МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РФ

Федеральное государственное бюджетное образовательное учреждение высшего образования
«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
вычислительной техники	МОиПЭВМ

Направление подготовки 09.03.04 «Программная инженерия»

БАКАЛАВРСКАЯ РАБОТА

на тему

Микросервис потоковой обработки событий на платформе Apache Kafka

Студент _____ Ершов Даниил Вадимович
(подпись, дата) (ФИО полностью)

Руководитель _____ Шибанов С.В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр _____ Такташкин Д.В.
(подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол заседания кафедры от _____№_____)

Заведующий кафедрой _____ Козлов А.Ю.
(подпись) (фамилия, инициалы)

Работа защищена с отметкой _____ (протокол заседания ГЭК от _____№____)

Секретарь ГЭК _____ Попова Н.А.
(подпись) (фамилия, инициалы)

Пенза, 2023

Федеральное государственное бюджетное образовательное учреждение высшего образования
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ПОЛИТЕХНИЧЕСКИЙ ИНСТИТУТ

Факультет	*Кафедра*
Вычислительной техники	МО и ПЭВМ

«Утверждаю»
Заведующий кафедрой
_____А.Ю. Козлов
«__»_____20__ г

ЗАДАНИЕ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
БАКАЛАВРА

1. Студент Ершов Даниил Вадимович гр. 19ВП1 факультета ВТ
<Фамилия, имя, отчество полностью>

2. Тема работы Микросервис потоковой обработки событий на платформе Apache Kafka

Тема утверждена приказом ПГУ №_____ от "___"_____ 20___

3. Руководитель работы к.т.н., доцент кафедры «МОиПЭВМ» Шибанов С.В.

4. Задание на работу (назначение разработки, исходные данные и т.п.)

1) Разработка архитектурного/технологического решения микросервиса потоковой обработки событий на платформе Apache Kafka
2) Среда выполнения: Linux/Windows
Языковые средства разработки: Java, Groovy, YAML, AVRO
Инструментальные средства разработки: Inteliji IDEA, Docker, Postman, Visual Topology, Offset Explorer, Draw.io.
Технологии разработки: ООП, АОП, Функциональный подход
3) Основные функции микросервиса потоковой обработки событий:
а) способность микросервиса оперировать потоком гетерогенных событий по мере его движения посредством непрерывных вычислений для реагировния без необходимости явного сбора или сохранения этих событий;
б) способность микросервиса анализировать входной поток простых событий с целью извлечения из него сложных событий;
в) обогащение событий метаданными;
г) маршрутизация событий.

5. Перечень подлежащих разработке вопросов

1) Анализ предметной области и постановка задачи

2) Проектирование

3) Реализация

4) Контроль качества программного обеспечения

5) Оформление пояснительной записки

6. Календарный график выполнения работы

№ п/п	Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1	*Анализ предметной области постановка задачи*	10%	12.04.2023–16.04.2023	
2	*Проектирование*	20%	17.04.2023–2.05.2023	
3	*Реализация*	30%	03.05.2022 – 03.06.2022	
4	*Контроль качества программного обеспечения*	20%	04.06.2022 – 08.06.2022	
5	*Оформление пояснительной записки*	20%	09.06.2023–17.06.2023	

Дата выдачи задания "22" декабря 2022

Руководитель бакалаврской работы _____

(подпись, дата)

Задание к исполнению принял студент _____

(подпись, дата)

Работу к защите допустить

Декан факультета _____ Л.Р. Фионова

(подпись, дата)

Реферат

Пояснительная записка содержит 92 листов, 31 рисунок, 1 таблица, 44 использованных источника, 4 приложений.

ПОТОКОВАЯ ОБРАБОТКА, CEP, РАСПРЕДЕЛЕННЫЕ СИСТЕМЫ, ТОПОЛОГИЯ ПРОЦЕССА, APACHE KAFKA, JAVA, ООП, АОП

Объектом разработки является часть сервиса потоковой обработки событий и исполнения активных правил – микросервис потоковой обработки событий.

Цель работы – создание подходящего архитектурного/технологического решения по созданию микросервиса потоковой обработки событий 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 микросервиса потоковой обработки событий на основе одного из функциональных сценариев обработки событий.

Содержание

Введение 7
1. Анализ предметной области и постановка задачи на разработку.. ……..9
1.1 Сервис потоковой обработки событий и исполнения активных правил 9
1.1.1 Общая концепция системы 9
1.1.2 CEP на практике 10
1.1.3 Назначение и требования сервиса потоковой обработки событий и исполнения активных правил. 11
1.1.4 Функциональные возможности сервиса потоковой обработки событий и исполнения активных правил 12
1.1.1 Структура метаданных сервиса потоковой обработки событий и исполнения активных правил 14
1.1.2 Архитектура сервиса потоковой обработки событий и исполнения активных правил 15
1.2 Постановка задачи на разработку программных средств 18
2 Проектирование программного обеспечения 18
2.1 Разработка требований к программным средствам 19
2.1.1 Нефункциональные требования к программным средствам 19
2.1.2 Функциональные требования к программным средствам 20
2.2 Разработка архитектуры программных средств 22
2.2.1 Выбор технологий разработки 24
2.3 Разработка сценария обработки событий. 28
3 Разработка программных средств микросервиса потоковой обработки событий. 37
3.1 Разработка программных средств потоковой обработки 37
3.2 Реализация программных средств распределенной шины сообщений 43
3.3 Тестирование программных средств 46
Заключение 49
Список использованных источников 50

Приложение А. Архитектура 55
Приложение Б. Конфигурация системы. 58
Приложение В. Тестирование. 66
Приложение Г. Кодовая база 75

Введение

В современном информационном мире проблема обработки данных становится важной и актуальной из-за стремительного роста объема генерируемой и собираемой информации. Источниками данных могут выступать социальные сети, различные web-системы, устройства Интернета вещей (IIoT), производственные предприятия, даже сам интернет и так далее. В качестве примера можно взять сферу IIoT, где автомобили, напичканные тысячами датчиков, могут генерировать впечатляющий объем данных, составляющий до нескольких терабайт в день или современную информационную банковскую сферу, где только за секунды могут совершаться сотни тысяч транзакций, в процессе производя колоссальный объем информации. Для таких объемов данных очень часто ставятся задачи по их обработке и анализу в реальном времени. Данная задача представляет тяжелый вызов и требует особых методов и подходов. Парадигма MapReduce [1], испытанная и широко применяемая на различных предприятиях, предоставляет упрощенный способ параллельной обработки таких огромных объемов данных на больших кластерах аппаратного оборудования, обеспечивая надежность и отказоустойчивость. Тем не менее, данная парадигма не является подходящей, так как она была специально разработана для пакетной обработки статических данных, где все входные массивы должны быть предварительно собраны в хранилище. Одним из наиболее возможных алгоритмических решений является обработка записей прямо в процессе их передачи из источника в приемник, данный подход называется потоковой обработкой данных [2]. Однако, для того чтобы добиться анализа такого потока, нужно реализовать так называемую комплексную обработку событий – CEP(Complex-Event-Processing).

Системы с таким подходом позволяют обнаруживать некоторые совпадения в общей транслируемой массе, анализировать и обрабатывать их по мере поступления в реальном времени, а также быстро на них реагировать.

Основной задачей данной работы будет рассмотрение алгоритмических, архитектурных и технологических подходов с определенными требованиями для разработки части системы – «Сервиса потоковой обработки событий и исполнения активных правил», отвечающей за обработку и выявления сложных событий в потоке данных, и в конечном итоге определение наиболее оптимального метода к ее программной реализации.

В главе 1 будет описано устройство всей системы, а также выдвинуты конкретные цели данной работы, в главе 2 будет описана версия архитектуры на разработку, глава 3 будет служить описанием разработки.

1 Анализ предметной области и постановка задачи на разработку

После введения данный раздел содержит раздел 1.1, который немного раскроет принципы CEP, а также введет архитектурный паттерн существующей системы. На основании этой информации будет выдвигаться постановка задачи на разработку в разделе 1.2.

1.1 Сервис потоковой обработки событий и исполнения активных правил

1.1.1 Общая концепция системы

Перед более детальным обзором системы стоит разобраться с ее ключевой концепцией работы – CEP [3, 4, 5].

Изначально весь поток входных данных представляется как очередность простых/атомарных/элементарных событий (Event), обработка которых осуществляется с привлечением стандартных CEP-механизмов, которые умеют выявлять и реагировать на так называемые сложные события (Complex Event), состоящие из простых.

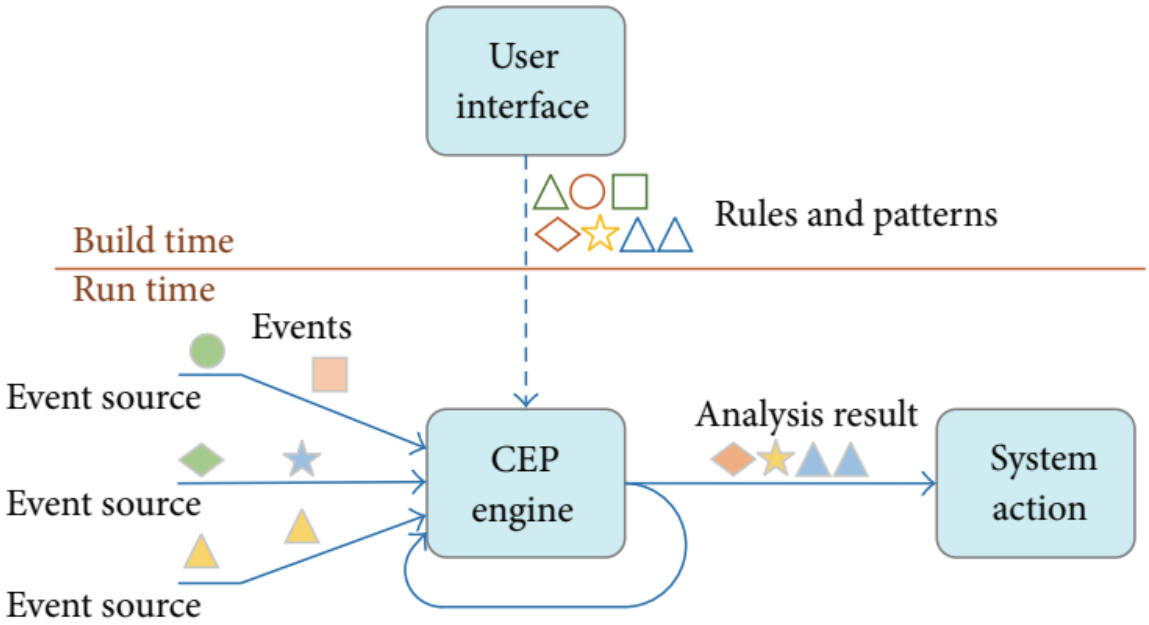

Рисунок 1 – Общая архитектура CEP-механизма

Модель CEP состоит из трех ключевых компонентов представленных на рисунке 1, которые должны выполняться последовательно.

Событийная часть (Event source). Часть, объявляющая, инициализирующая события, которые должны отслеживаться системой.

Часть условия (Rules and Patterns). Представляет собой логический блок, в котором устанавливаются заранее заданные условия и критерии. При их выполнении, соответствующие действия и операции активируются автоматически.

Часть действий (System Action). Состоит из разнообразных операций и механизмов взаимодействия с наблюдаемой системой. Это включает в себя выполнение активных правил – правил реагирования системы на события, передачу данных или активацию соответствующих процессов.

В данной работе основное внимание будет уделяться событийной части модели CEP по выявлению сложных событий, так как именно она будет применяться ко входному потоку данных.

1.1.2 CEP на практике

Способ реализации CEP модели на практике является не таким тривиальным. Главным аспектом CEP является выявление общих паттернов событий и их логической связанности. Для достижения этой цели недостаточно анализировать события поодиночке, необходимо сохранять некое промежуточное состояние, которое позволяет установить контекст и взаимосвязи между ними. Чаще всего для реализации данной задачи используются алгоритмы накопления событий в буфере, что является хорошим решением для небольшого объема данных, или складирование их в каком-либо высокоемком хранилище на время обработки, например базе данных (БД). Также, для реализации промежуточного состояния применяются очереди сообщений, что является более лучшим решением для обработки потоков в контексте скорости.

В последствии, хранение данных в очередях привело к созданию концепций по оптимизации, например такой как «БД наизнанку» [6], полностью переворачивающей роли во взаимодействии БД и очередей. Если говорить коротко, то очередь становится системой, которая начинает аккумулировать все данные, являясь источником истины [7, 8] для приложения и поставляя ему информацию, в то время как БД отводится роль лишь временного хранилища для производимых операций.

Данная концепция позволила хранить данные состояний обработки событий наиболее близко к источнику их приема, что особенно сильно сказалось на оптимизации алгоритма модели CEP.

1.1.3 Назначение и требования сервиса потоковой обработки событий и исполнения активных правил

Сервис потоковой обработки событий, реализующий механизм активных правил, принимает на себя функцию высоконагруженной платформы, которая принимает и обрабатывает непрерывный поток событий, поступающий от внешних наблюдаемых систем в реальном или близко к реальному времени. Платформа имеет потенциал развития с разной направленностью от IoT сферы [9, 10] до не менее популярной сферы анализа BigData [11]. В настоящий момент курс разработки направлен по пути обслуживания данных web-приложений в виду более простой и известной специфики, для изменения области разработки должно будет проводиться частичное изменение архитектуры.

Далее представлены базовые нефункциональные требования, которым отвечает система.

Надежность – необходимость в том, чтобы система сохраняла свою работоспособность и выполняла все требуемые функции с достаточным уровнем производительности даже при возникновении неблагоприятных обстоятельств. Эти неблагоприятные обстоятельства могут включать в себя сбои в аппаратной части системы, программные ошибки или неправильные действия со стороны

пользователя. Важно, чтобы система продолжала функционировать надежно и без сбоев, несмотря на такие ситуации, и обеспечивала выполнение всех необходимых функций с оптимальной производительностью.

Масштабируемость – необходимость в эффективных механизмах для обнаружения и решения проблем, которые могут возникнуть при увеличении объемов данных, трафика, обращений в секунду – requests-per-seconds (RPS) или сложности системы. На данный момент объем данных ограничивается пределами одной машины.

Удобство сопровождения – обеспечение удобной и эффективной работы с системой множеству людей, от разработчиков и до администраторов.

Далее представлены базовые функциональные требования, которым отвечает система.

Способность оперировать бесконечным потоком гетерогенных (но логически связных событий) по мере его движения посредством непрерывных вычислений для реагирования без необходимости сбора или сохранения этих событий;

Система должна, по мимо простого анализа входного потока гетерогенных данных обладать способностью извлекать более высокие уровни абстракции из этих самых данных для понимания сложных ситуаций и реакции на них либо составления каких-либо последующих прогнозов;

Система должна определенным образом реагировать на идентифицированные события по заранее заданному сценарию.

Данные требования отражаются на архитектуре системы и также будут учитываться в работе.

1.1.4 Функциональные возможности сервиса потоковой обработки событий и исполнения активных правил

Перейдем к функциональным возможностям.

Для представления о системе и понимания конкретного применения CEP нужно взглянуть на рисунок 2, где представлена диаграмма вариантов использования [12] сервиса.

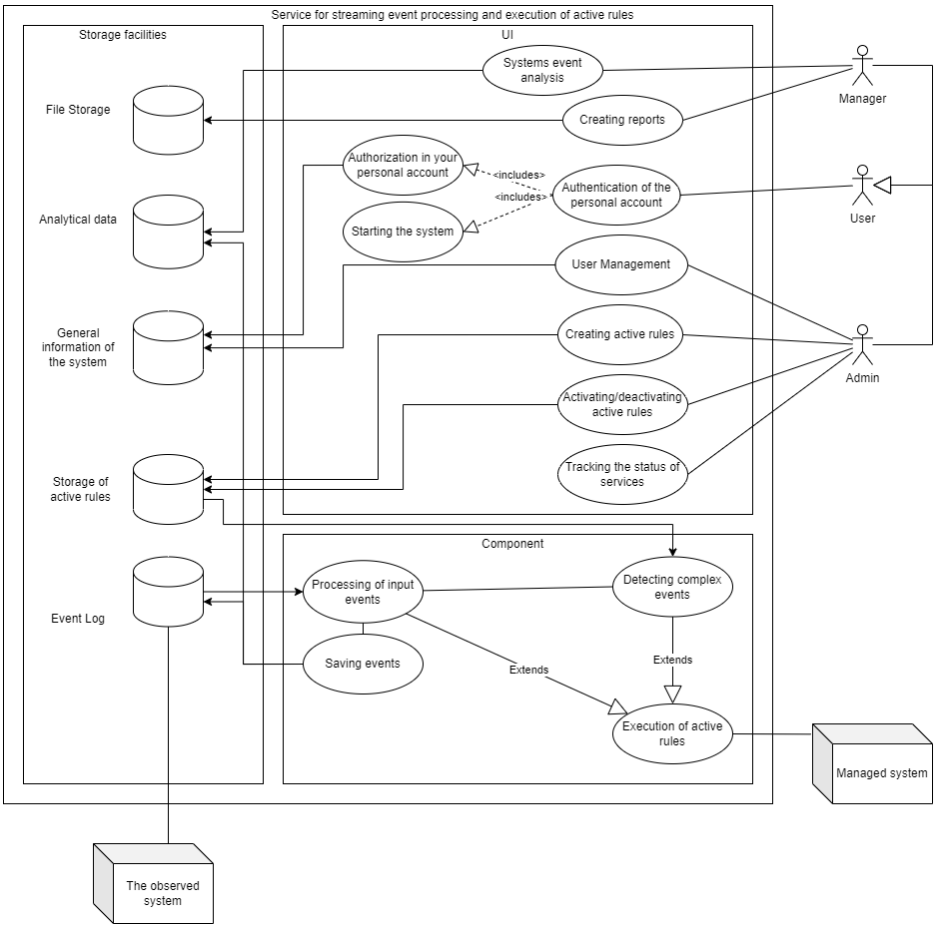

Рисунок 2 – Диаграмма вариантов использования сервиса потоковой обработки событий

Функциональные возможности сервиса, представленные на диаграмме:

– прием и сохранение (Saving events) событий от отслеживаемых объектов в распределенной очереди/журнале (EventLog) с целью их обработки;

– потоковая обработка входных событий (Processing of input events), обнаружение сложных и агрегированных событий на основе предопределенных шаблонов (Detecting complex event);

– исполнение активных правил (Execution of active rules);

– регистрация, аутентификация/авторизация пользователя в личном кабинете (User management);

– создание активных правил через личный кабинет пользователя (Creating active rules);

– управление активными правилами, их активация/деактивация, корректировка (Activating/Deactivating active rules);

– отслеживание жизненного цикла и состояния сервиса (Tracking the status of servic);

– создание отчетов (Creating reports) аналитиками по прогнозам, произведенным над данными (Systems event analysis).

Для более полного представления рекомендуется обратиться к имеющейся спецификации [13].

1.1.1 Структура метаданных сервиса потоковой обработки событий и исполнения активных правил

В сервисе потоковой обработки событий, метаданные представляют собой данные, которые описывают свойства и характеристики событий или активных правил. Эти данные включают информацию, такую как тип события, временная метка, источник и другую дополнительную информацию, которая имеет отношение к обработке события или выполнению правила.

Структура метаданных является центральной моделью, определяющей и классифицирующей приходящие события. Концептуальная структура метаданных сервиса на данный момент представлена на рисунке 3.

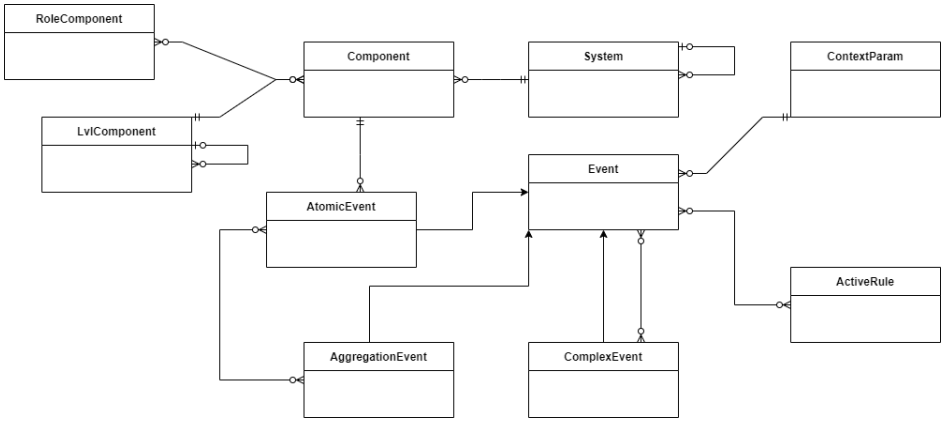

Рисунок 3 – Концептуальная модель репозитория метаданных

Далее представлены основные идентификационные сущности системы.

Атомарное событие (Atomic Event). Классификация возникающего события в результате получения данных от внешних систем мониторинга в определенный момент времени и обусловлено действиями пользователей, приложений или изменениями данных.

Сложное событие. Представляет собой и/или другими сложными событиями. Сложные события могут быть агрегированными или составными, в зависимости от их характера.

Агрегированное событие (Aggregated Event). Классификация вычисляемого события, которое система обнаруживает на основе различных операций агрегации над атомарными.

Составное событие (Complex Event). Является комбинацией атомарных и/или сложных событий, учитывая ограничения, логические связи и последовательность во времени.

За подробной структурой репозитория метаданных стоит обратиться к спецификации [14].

1.1.2 Архитектура сервиса потоковой обработки событий и исполнения активных правил

Событийная природа системы и удержание высокой нагрузки, вследствие обработки большого количества данных, а также указанные требования в ранее приведенном разделе 1.1 требуют создание архитектуры со свойствами распределенности [15]. Выбор был сделан в пользу базового микросервисного подхода [16] и его паттернов [17, 18], использующего событийно-ориентированное взаимодействие – Event-Driven (EDA) [19] на основе очереди сообщений [20].

EDA, как уже было сказано, подходит системе по сущности происходящего процесса – обработке событий [21]. 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 [22] на рисунке 4. Также стоит упомянуть, что система постоянно развивается и в будущем некоторые компоненты могут быть убраны или изменены.

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

Микросервисы потоковой обработки событий(mcs-cep) и модуль исполнения активных правил(module-active-rules) являются ядром системы и фактически составляют единое целое. Микросервис CEP обрабатывает простые события в реальном времени, обогащает их метаданными [23], группирует/агрегирует/соединяет в абстракции более высокого уровня и передает их в шину для последующей маршрутизации. Модуль исполнения активных правил отвечает за исполнения активных правил. Поглощая входящие события всех видов, данный компонент выполняет соответствующую логику реакции, которая была заранее программно задана. 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(msc-s3/mcs-print) для возможного анализа событий data-инженером.

Отдельного упоминания достойно получение ядром системы значений метаданных для обогащения. Хранилищем метаданных является центральный

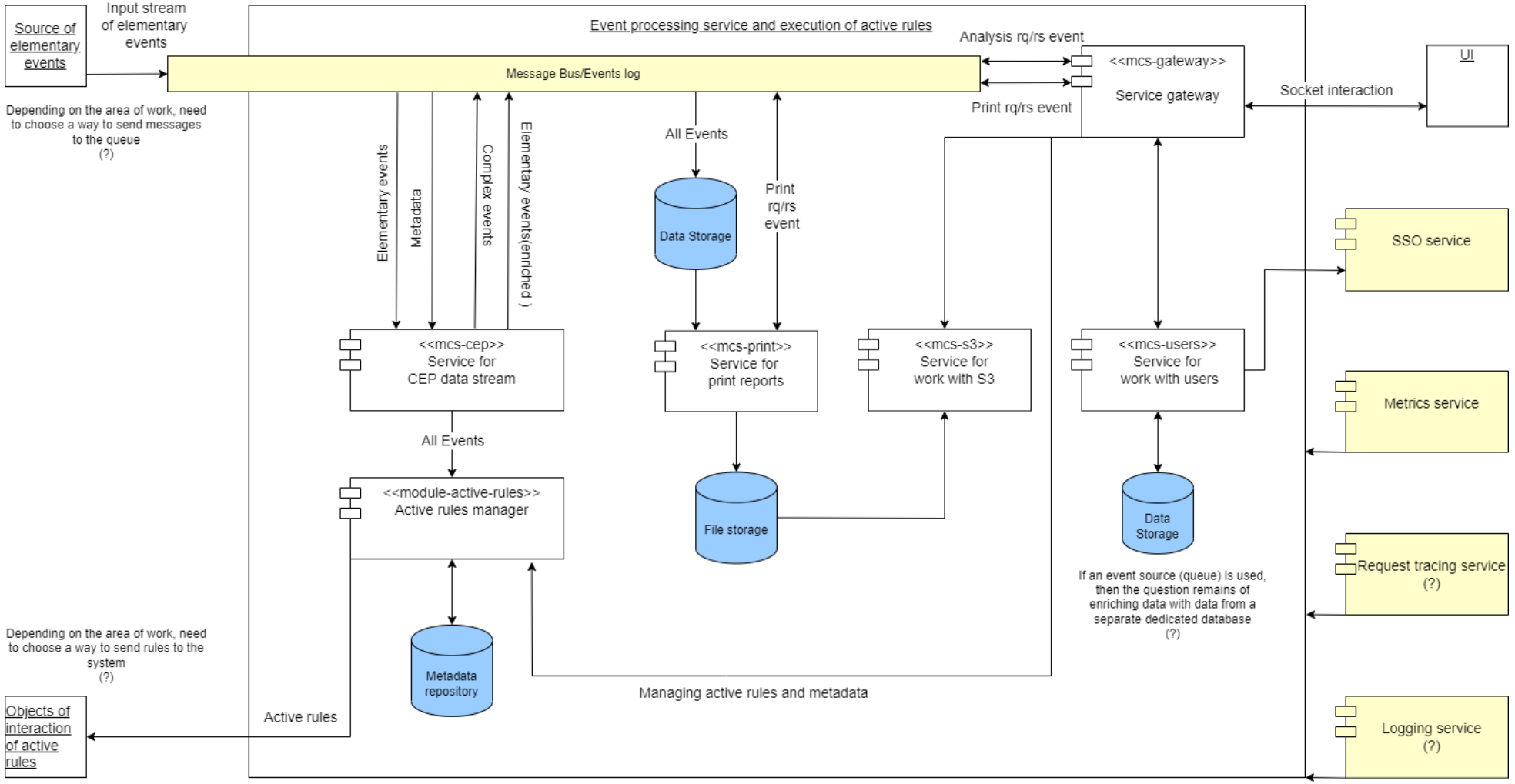

Рисунок 4 – Верхнеуровневая диаграмма компонентов С2 сервиса потоковой обработки событий и исполнения активных правил

репозиторий (metadata repository), который содержит спецификации событий и активных правил, важные промежуточные результаты работы, правила обработки событий и формирования управляющих воздействий, результаты анализа событий и правил и многое другое.

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 [24] архитектуре. Для взаимодействия с клиентскими приложениями, конфигурации сервиса и управлением активных правил имеется пользовательский сервис(mcs-users).

1.2 Постановка задачи на разработку программных средств

Данная работа будет преследовать цель по созданию подходящего архитектурного/технологического решения микросервиса потоковой обработки событий – части сервиса потоковой обработки событий и исполнения активных правил, а также настройке шины данных. Общее решение должно обеспечивать надежное и быстрое выявления сложных событий по модели CEP во входящем потоке данных. Далее представлены основные цели разработки.

Организация способности оперировать бесконечным потоком гетерогенных событий по мере его движения, посредством непрерывных вычислений, для реагирования без необходимости сбора или сохранения этих событий.

Организация доступных механизмов анализа входного потока простых событий, с извлечением из него сложных событий.

Организация надежности и масштабируемости шины и самого микросервиса.

Также стоит упомянуть, что цели не включают в себя разработку модуля исполнения активных правил.

2 Проектирование программного обеспечения

В данной главе будет рассмотрено создание устойчивой архитектуры микросервиса потоковой обработки событий, подходящей для последующей разработки в данной области. В разделе 2.1 будут специфицированы требования к программным средствам, в разделе 2.2 будет установлена концептуальная архитектура разрабатываемого микросервиса, раздел 2.3 будет один из возможных сценариев потоковой обработки.

2.1 Разработка требований к программным средствам

2.1.1 Нефункциональные требования к программным средствам

После определения целей разработки в главе 1 нужно указать требования программного обеспечения (ПО) для микросервиса потоковой обработки событий. Нефункциональные требования, представленные в главе 1 для всего сервиса, практически полностью будут дублироваться и для разрабатываемого микросервиса. Далее идут упомянутые требования.

Требование надежности – способность микросервиса сохранять непрерывную работоспособность и оптимальную производительность вне зависимости от неблагоприятных обстоятельств, таких как аппаратные сбои, программные ошибки или неправильные пользовательские действия, микросервис должен продолжать функционировать стабильно и исполнять все необходимые функции без сбоев.

Требование масштабируемость – способность эффективно управлять ростом объемов данных, трафика и сложности системы. Данные ограничены хранением в пределах одной машины.

Удобство сопровождения – обеспечение удобной и эффективной разработки продукта.

Данные нефункциональные требования будут влиять на принятие тех или иных технических решений на разработку микросервиса потоковой обработки.

2.1.2 Функциональные требования к программным средствам

Исходя из целей разработки, указанных в главе 1, можно выделить следующие функциональные требования:

– способность микросервиса оперировать бесконечным потоком гетерогенных событий по мере его движения посредством непрерывных вычислений для реагирования без необходимости явного сбора или сохранения этих событий (Processing of input events);

– способность микросервиса анализировать входной поток простых событий, с извлечением из него сложных событий (Detecting complex events);

– обогащение событий метаданными (Enriching events);

– маршрутизация событий по всей системе (Routing events);

– настройка отказоустойчивости шины сообщений и самого микросервиса (Configure fault tolerance);

– возможность надежного и доступного хранения событий. Данный пункт подразумевает под собой использование репликации (Configure fault tolerance);

– настройка гарантированной записи и обработки событий (Configure event delivery guarantee);

– настройка консистентности записи и обработки событий;

– возможность наращивать пропускную способность системы и шины благодаря настройкам масштабирования (Control the throughput);

– должны предоставляться механизмы по отслеживанию различных метрик приложения с целью его управления (Providing metrics);

– организация допустимой задержки по обработке входных событий.

– должны предоставляться механизмы по ведению историчности действий системы (Logging data).

Более наглядно требования представлены на диаграмме вариантов использования рисунка 5.

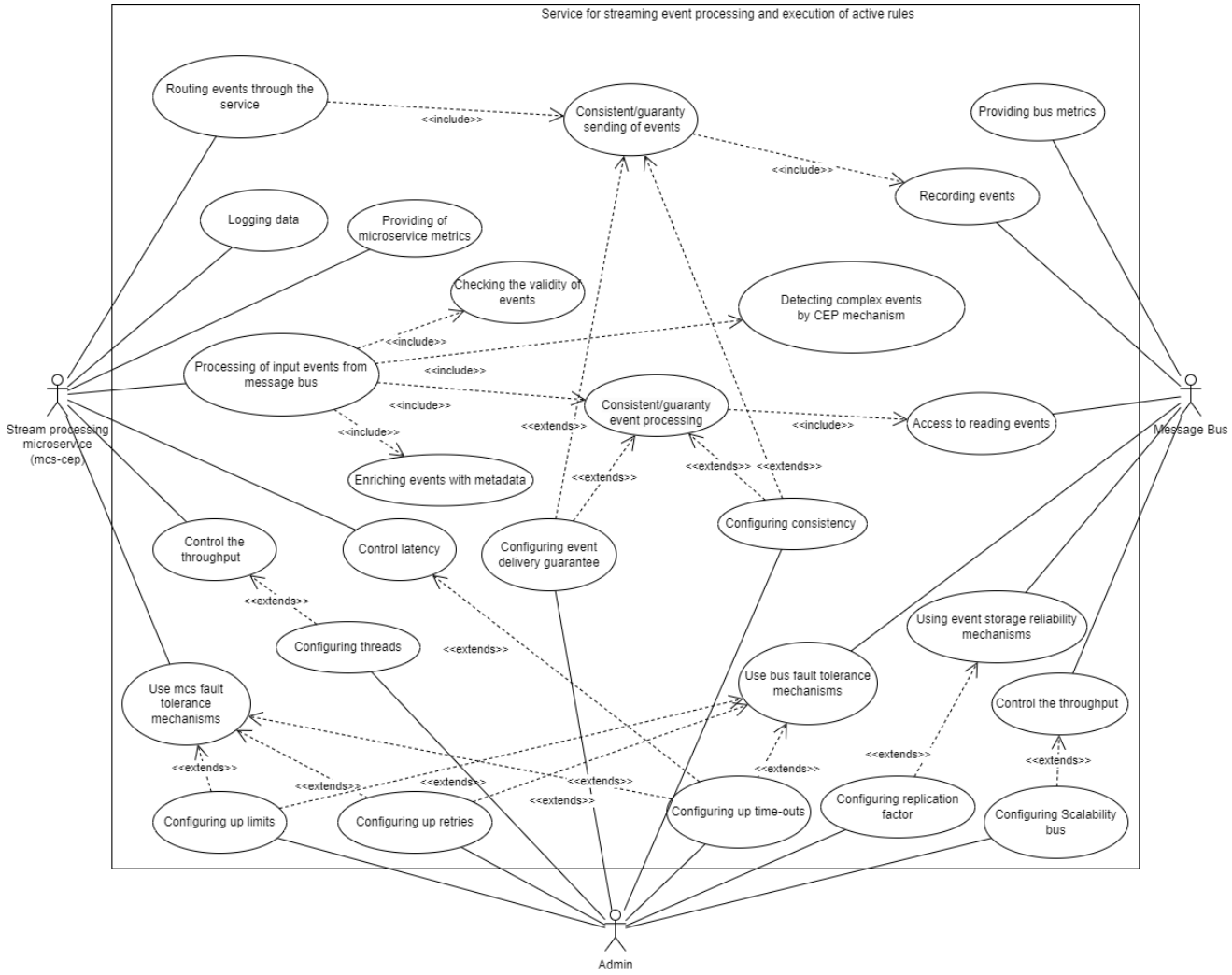

Рисунок 5 – Диаграмма вариантов использования микросервиса потоковой обработки событий

Пункт консистентности стоит аккуратно учитывать в контексте возможности распределенности системы хранения, так как CAP теорема [25] ставит некоторые ограничения (Configure consistency).

Пункт организации задержки требует пояснения, так как в распределенных системах невозможно достичь нулевой задержки, как минимум из-за неопределенности взаимодействия по сети. Но есть способы по увеличению данного показателя. В качестве эталона/допустимой нормы можно привести директиву MiFID II в обработке транзакций [26], к которой стоит стремиться (Control latency).

По диаграмме существует 3 типа акторов: микросервис потоковой обработки событий, шина и администратор. Администратор может управлять настройкой всех ключевых параметров до старта приложения. Шина сообщений представлена в упрощенном виде со способностью записывать и давать доступ на чтение.

2.2 Разработка архитектуры программных средств

Как было упомянуто в главе 1, весь сервис представляет собой распределенную архитектуру, где внутренние сервисы общаются между собой посредством передачи сообщений в шину. Исходя из того, что весь сервис был заточен на web-взаимодействие, было принято решение вести разработку микросервиса с использованием аналогичного подхода. В стилистике данной архитектуры существуют понятия клиентской и серверной частей приложения. Клиентская часть, или Frontend, представляет собой пользовательский интерфейс, который обеспечивает взаимодействие с пользователем. Серверная часть, или Backend, отвечает за обработку запросов, бизнес-логику и взаимодействие с другими сервисами. Учитывая, что разрабатываемое ПО представляет собой микросервис потоковой обработки событий, его логику можно полностью отнести к серверной части приложения.

Проектируемая архитектура должная придерживаться принципов и политик «чистой архитектуры» [27], описанных Робертом Мартином. Отражение

данных принципов несет в себе слоеная архитектура [28, 29], которая будет использоваться при разработке микросервиса.

Приложение может быть разделено на два слоя, показанных на рисунке 6, вместо стандартных трех, так как в потоковой обработке взаимодействие с пользователем отсутствует.

Слой работы к данными (Data Layer) – слой, отвечающий за обработку данных и реализацию CEP алгоритмов.

Слой бизнес-логики (Business Layer) – набор компонентов, которые реализуют логику основного функционала приложения. В данном случае такой логикой может являться реализация активации CEP правил по реагированию на события.

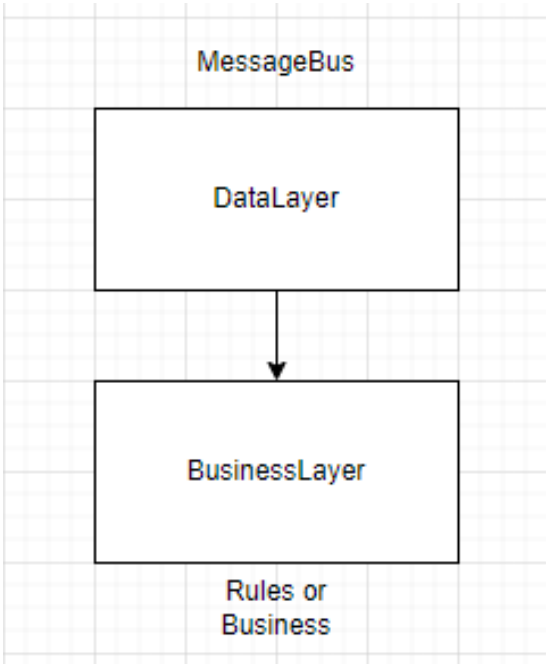

Рисунок 6 – Схема слоеной архитектуры

Такое разделение на слои способствует проведению архитектурных горизонтальных границ [27], и, следовательно, помогает добиться гибкости в изменениях, а значит и в разработке.

2.2.1 Выбор технологий разработки

2.2.1.1 Центральная шина

Для реализации поставленных целей по разработке микросервиса прежде всего стоит рассмотреть варианты технологий, которые будут составлять реализацию шины данных. На сегодняшний день существует несколько технологий, которые способны удовлетворить указанным требованиям сервиса, о надежности и масштабируемости.

Apache Kafka [30]. Представляет собой распределенную платформу передачи событий с открытым исходным кодом, являющуюся одним из наиболее активных проектов Apache Software Foundation. Она разработана в качестве реплицируемого, распределенного и постоянного журнала фиксации, предназначенного для управления событиями в микросервисах или крупномасштабных приложениях потоковой обработки. Взаимодействие клиентов осуществляется непосредственно с кластером брокеров, который постоянно осуществляет чтение и запись событий в базовую файловую систему. Кроме того, события автоматически реплицируются внутри кластера синхронно или асинхронно, обеспечивая отказоустойчивость и высокую доступность.

Apache Pulsar [31]. Является распределенной системой обмена сообщениями с открытым исходным кодом, обладающей возможностью потоковой передачи событий. Ее архитектура основана на безопасных брокерах, которые соединены с отдельным слоем экземпляров BookKeeper (другой системой хранения распределенных данных). Эти экземпляры фактически осуществляют чтение, запись и долгосрочное хранение, а при необходимости репликацию сообщений.

RabbitMQ [32]. Представляет собой промежуточное open-source ПО для обмена сообщениями на основе стандарта AMQP для организации очередей с низкой задержкой. Поэтому скорость работы и пропускная способность этого фреймворка зависит от обмена сообщениями как на стороне производителя, так и на уровне очередей, связанных с ними на стороне потребителя.

Исходя из краткого описания, информации в документации по всем трем технологиям на данный момент, а также на основании технического сравнения компании Confluent [33] можно сделать некоторые выводы.

Масштабируемость. Pulsar и Kafka оба обладают высоким уровнем масштабируемости, позволяя добавлять больше брокеров и внутренних разделов, чтобы обрабатывать больший объем данных и поддерживать большое количество подключенных клиентов. Однако, для полноценной работоспособности Pulsar необходима технология BookKeeper, что будет являться лишними бюджетозатратами во время разворачивания полной инфраструктуры. RabbitMQ же вообще не способен обеспечить экстремальную масштабируемость. Также благодаря данной масштабируемости Kafka обеспечивает самую высокую пропускную способность среди всех систем и практически самую наименьшую задержку. Противопоставить что-то здесь может только RabbitMQ, но для этого нужно значительно уменьшать нагрузку.

Надежность. Pulsar, Kafka и RabbitMQ предлагают механизмы репликации и большой уровень избыточности для обеспечения надежности и отказоустойчивости. В данном пункте все системы держат паритет, несмотря на некоторые нюансы. Но среди всех технологий в отрицательную сторону выделяется Rabbit, так как у него есть проблемы с надежным хранением данных.

Отвлеченные факторы. Pulsar – технология, подающая большие надежды, но на данном этапе она только развивается и не обладает той обширной поддержкой IT-сообщества, которой обладает Kafka. RabbitMQ, технология, которая в изначальной своей сути работает по протоколу очередей AMQP и фактически не создана для обеспечения высокой доступности, но постепенно пытается развеять этот миф. Большим минусом RabbitMQ является тот факт, что внутреннее устройство написано на языке Erlang, который хоть и был создан для работы с параллельностью, но недостаточно популярен среди разработчиков.

Можно утверждать, что на данный момент проект Apache является Kafka является одним из самых зрелых решений на рынке. Данная технология также

удовлетворяет всем выдвинутым функциональным/нефункциональным требованиям и с легкостью позволяет провести архитектурные границы для абстрагирования своей функциональности. Apache Kafka по праву занимает свое место среди существующих архитектурных решений. Для дальнейшего прочтения работы настоятельного рекомендуется ознакомится с документацией по Apache Kafka [30].

2.2.1.2 Серверная часть и потоковая обработка

Последним технологическим решением на разработку поверх выбранного фреймворка является выбор технологии потоковой обработки. На данный момент существует всего несколько достойных кандидатов:

Apache Storm [34]. Фреймворк с открытым исходным кодом для распределенных потоковых вычислений в реальном времени, разработанный на языке программирования Clojure. Большим минусом для обработки данных в реальном времени данной технологией является обработка пакетами, что может тормозить весь процесс;

SparkStreaming [35]. Целостная вычислительная система с набором библиотек для параллельной обработки данных на кластерах компьютеров. На данный момент Spark считается самым активно разрабатываемым средством с открытым кодом для решения подобных задач, что позволяет ему быть полезным инструментом для любого разработчика. Большим минусом данной технологии является отсутствие работы с хранилищами состояний.

Apache Flink [36]. Flink это фреймворк и движок для вычислений с использованием состояний над неограниченными и ограниченными потоками данных. Flink был разработан для работы во всех распространенных кластерных средах, выполняя вычисления с in-memory скоростью на любом масштабе данных.

KafkaStreams [37]. Kafka Streams, в отличие от других фреймворков потоковой передачи, представляет собой легковесную библиотеку. Она полезна для потоковой передачи данных из Kafka, выполнения преобразования и

последующей отправки обратно в Kafka. Технология может быть хорошо интегрирована с любым приложением и будет работать из «коробки».

Samza [38]. Фреймворк Samza очень похож на Kafka Streams. Обе технологии были разработаны одними и теми же инженерами, которые привнесли Samza в инфраструктуру LinkedIn, а затем основали компанию Confluent, где они продолжили работу над Kafka Streams. Однако, следует отметить, что Samza представляет собой масштабируемую версию Kafka Streams. В отличие от Kafka Streams, которая представляет собой библиотеку, предназначенную для разработки для одного сервиса, Samza является полноценным кластерным решением, обеспечивая распределенную обработку данных.

Исходя из приведенной выше информации, а также прочтенной документации и изучения сравнений [39, 40], можно сделать вывод о том, что наилучшим решением на данную разработку будет выбор технологии Kafka Streams. KafkaStreams, что немаловажно, разработан поверх Apache Kafka и представляет собой удобную и легкую библиотеку для разработки потоковых приложений и работы с состояниями, в то время как остальные технологии, такие как Apache Flink, Apache Storm и Apache Spark, являются полноценными фреймворками для распределенных систем, предоставляющими более широкий набор инструментов и возможностей для обработки потоков данных, что для решаемой задачи является избыточным. Также данные фреймворки несут в себе бОльшую архитектурную зависимость.

Для дальнейшего прочтения работы настоятельного рекомендуется ознакомится с документацией по Kafka Streams [37].

Выбор языка программирования и фреймворка делался по остаточному принципу. Так как Kafka преимущественно написана на языках программирования Java и Scala, было решено выбрать первый из них, так как у автора работы имеется опыт профессионального программирования на Java.

Поддержка фреймворка также могла бы облегчить выполняемую задачу, беря на себя часть работы. Выбор подходящей технологии для выполнения текущей задачи является сложным процессом, требующим соблюдение архитектурного принципа независимости разработки. На данный момент практически единственным достаточно развитым решением в нише web-разработки с поддержкой JVM-языков (Java Virtual Machine) был SpringBoot [41]. Данный фреймворк позволяет пользоваться такими паттернами как инверсия управления и внедрение зависимостей при помощи AOP-подхода, что является одним из важных факторов. AOP подход основан на паттернах рефлексии и позволяет пользоваться возможностями фреймворка без явных архитектурных зависимостей.

2.3 Разработка сценария обработки событий.

Данный раздел будет преследовать цель применения описанной архитектуры на практике, используя эфемерную модель данных и созданную для нее топологию процесса обработки данных, с целью показать результат и применимость выбранных решений на практике.

2.3.1 Проектирование модели данных

Модель данных представлена на ER-диаграмме логического уровня рисунка 7, остальные типы опущены для краткости. Участвующая в сценарии модель данных представляет собой часть модели, взятой из общей модели метаданных репозитория, указанной в главе 1 и представляет основные сущности репозитория:

– метаданные атомарного события (Atomic Event Metadata). Реализация сущности репозитория метаданных как Atomic Event;

– метаданные агрегированного события (Aggregated Event Metadata). Реализация сущности репозитория метаданных как Aggregated Event;

– метаданные комплексного события (Complex Event Metadata). Реализация сущности репозитория метаданных как Complex Event;

– атомарное событие (Atomic Event). Неидентифицированное(сырое) входное событие системы;

– агрегированное событие (Aggregated Event). Неидентифицированное(сырое) сложное событие, построенное на основе атомарных;

– составное событие (Complex Event). Неидентифицированное(сырое) сложное событие, представляющееся как комбинация элементарных и/или сложных событий.

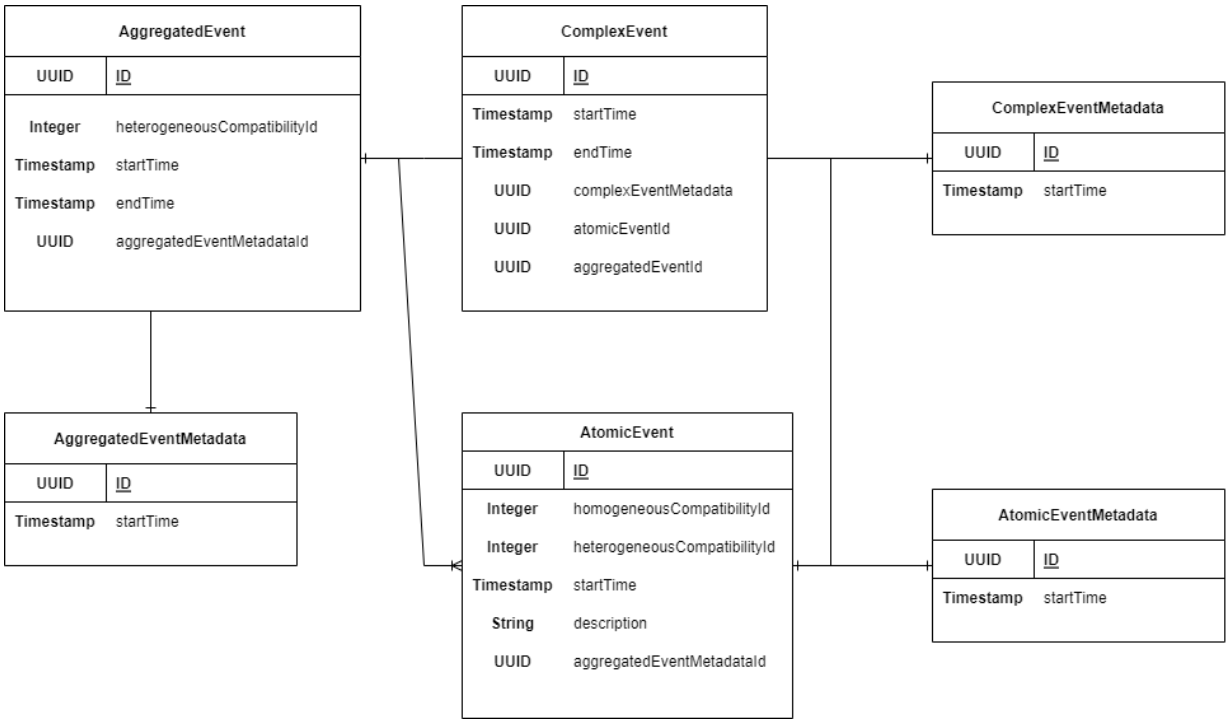

Рисунок 7 – Логическая модель данных сценария обработки событий

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

2.3.2 Сценарий обработки событий

Сценарий обработки события будет представлен в виде топологии процесса. Топология процесса – это способ представления потока данных в виде ориентированного ациклического графа – Directed Acyclic Graph (DAG), что

является одним из методов парадигмы программирования потоков данных [42]. Основная топология сценария обработки событий будет включать в себя обогащение входящих событий Atomic Event, их агрегация по ключам совместности и создание на основе агрегатов события Aggregated Event. Дальнейшая обработка базируется на слиянии объектов Atomic Event и Aggregated Event в событие Complex Event.

На рисунках далее представлены графы, которые являются реализацией разрабатываемого процесса, где данные проходят через набор узлов обработчиков (они обозначены прямоугольниками), а соединяющие их линии представляют собой потоки входных и выходных данных (которые передаются от одного обработчика к другому). Представленные графы являются частями общей топологии, но для удобства представления будут разделены на суб-топологии (части общей топологии) по типам обрабатываемых событий и производимым операциям. Топологии простого хранения в отрыве от операций учитываться не будут, например такие как AtomicEventMetadata на рисунке 8.

Все графовые представления были созданы с помощью инструмента Topology Visualizer, благодаря которому можно получить топологию процесса обработки KafkaStreams в виде набора инструкций и превратить ее в диаграмму.

Топология процесса обработки Atomic Event (рисунок 8) 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.

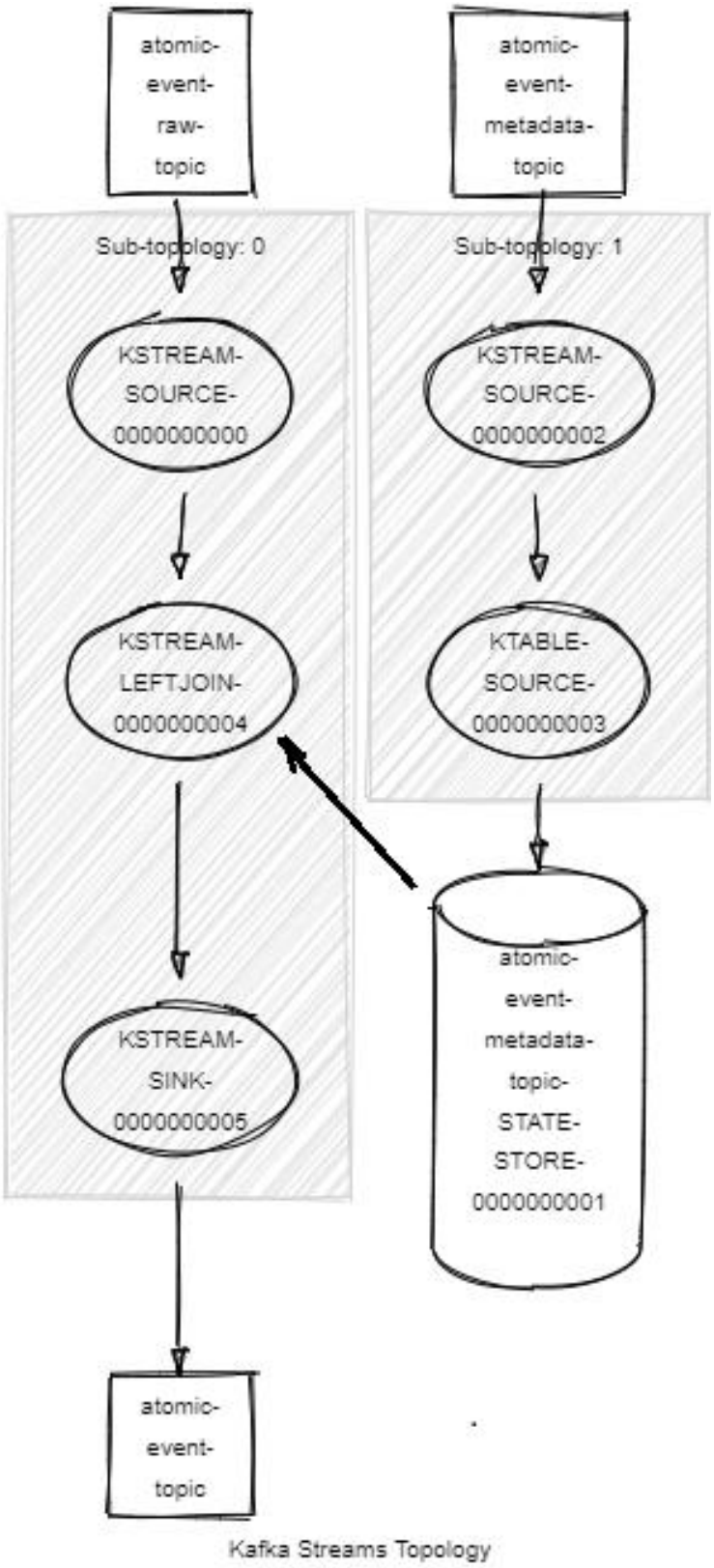

Рисунок 8 – Топология процесса обработки событий типа AtomicEvent

Топология процесса обработки события Aggregated Event представлена двумя суб-топологиями. На рисунке 9 представлена первая суб-топология,

которая включает в себя топологию обработки Atomic Event. Вторая суб-топология (рисунок 10) представляет из себя агрегацию обогащенных событий типа Atomic Event по ключу совместности (homogeneousCompatibilityId). Полученные события имеют промежуточный тип необогащенного события Aggregated Event Raw. Вторая часть второй суб-топологии представляет из себя обогащение событий типа Aggregated Event Raw по средствам соединения их с событиями Aggregated Event Metadata. Полученные события имеют тип Aggregated Event. Также важным упоминанием является создание хранимого состояния в момент агрегирования события.

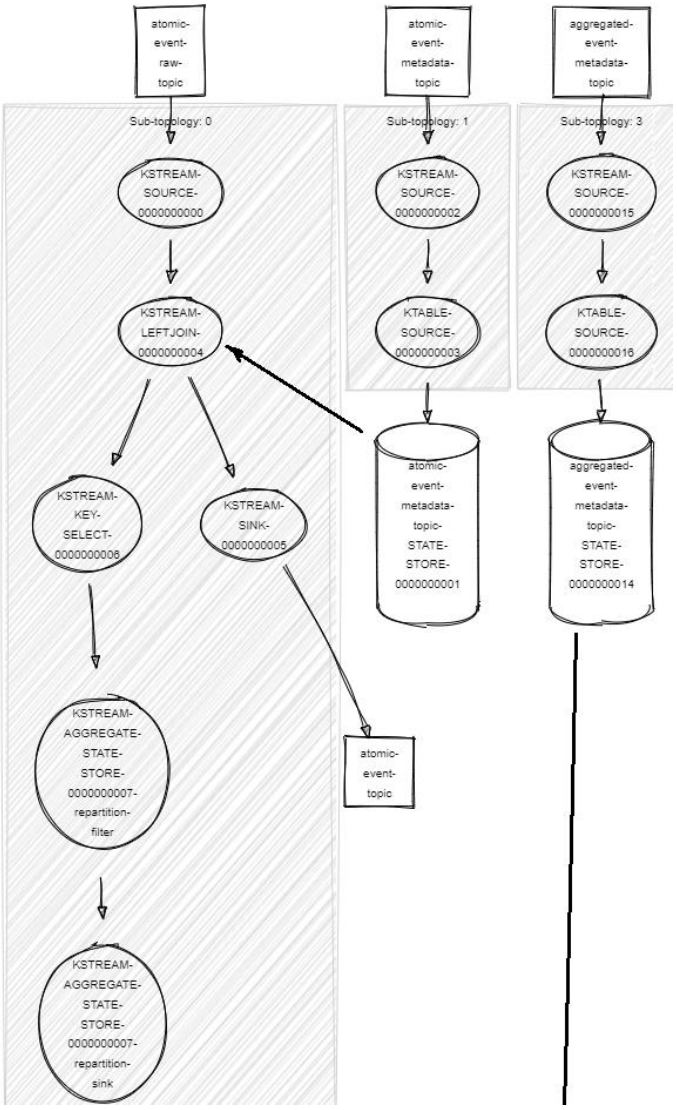

Рисунок 9 – Суб-топология процесса обработки событий типа AggregatedEvent (обогащение AtomicEvent)

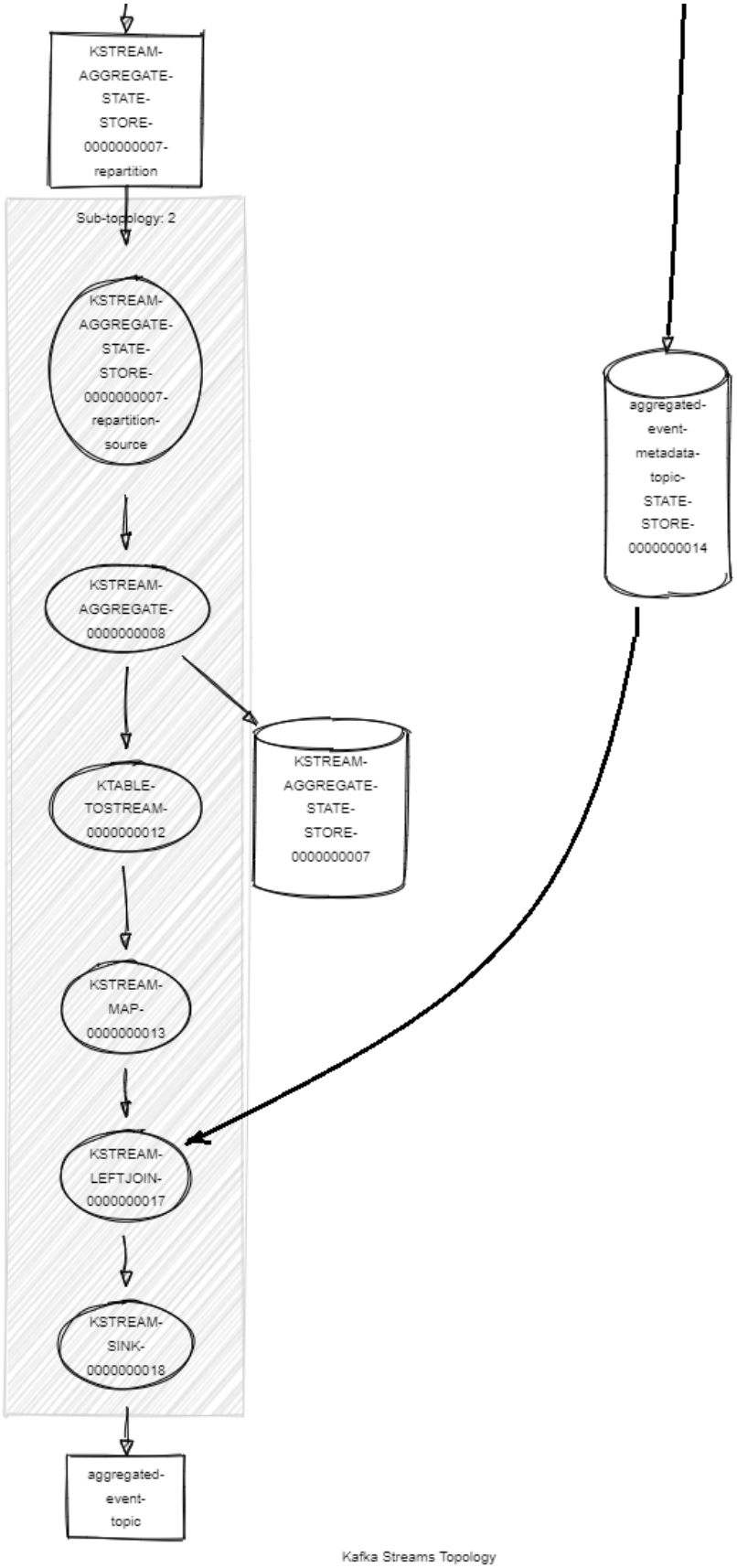

Рисунок 10 – Суб-топология процесса обработки событий типа AggregatedEvent (операции агрегирования и обогащения AggregatedEvent)

Топология процесса обработки события Complex Event включает в себя все предыдущие указанные топологии. Всего суб-топологий по обработке Complex Event будет четыре. Первые две суб-топологии аналогичны обработке Atomic Event и Aggregated Event, третья топология (рисунок 11) представляет из себя секционирование ключей (перераспределение ключей). Четвертая суб-топология (рисунок 12) будет объединять события Atomic Event и Aggregated Event за счет временных окон.

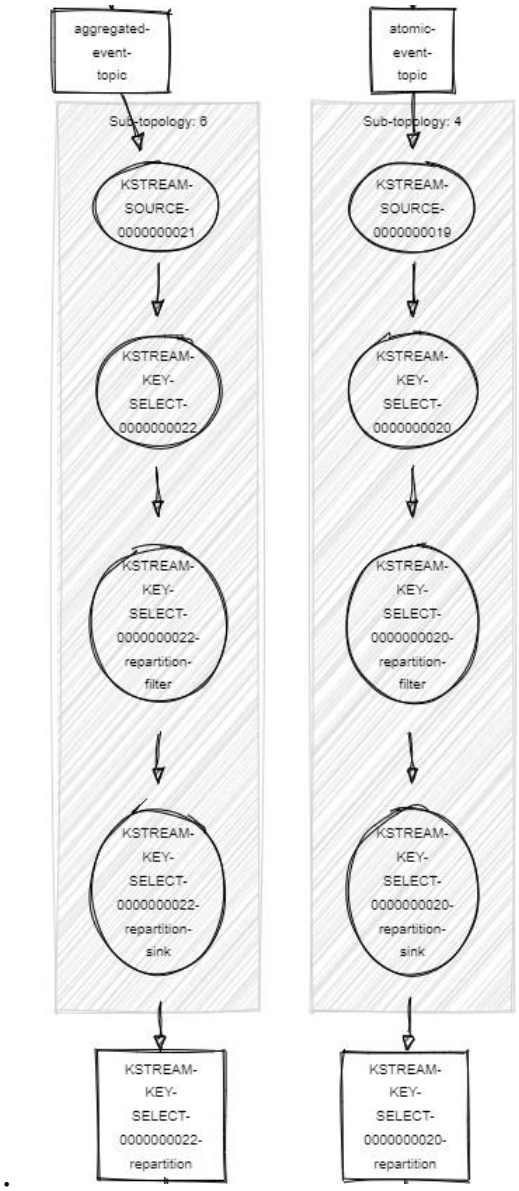

Рисунок 11 – Суб-топология процесса обработки Complex Event (секционирование событий Atomic Event и Aggregated Event)

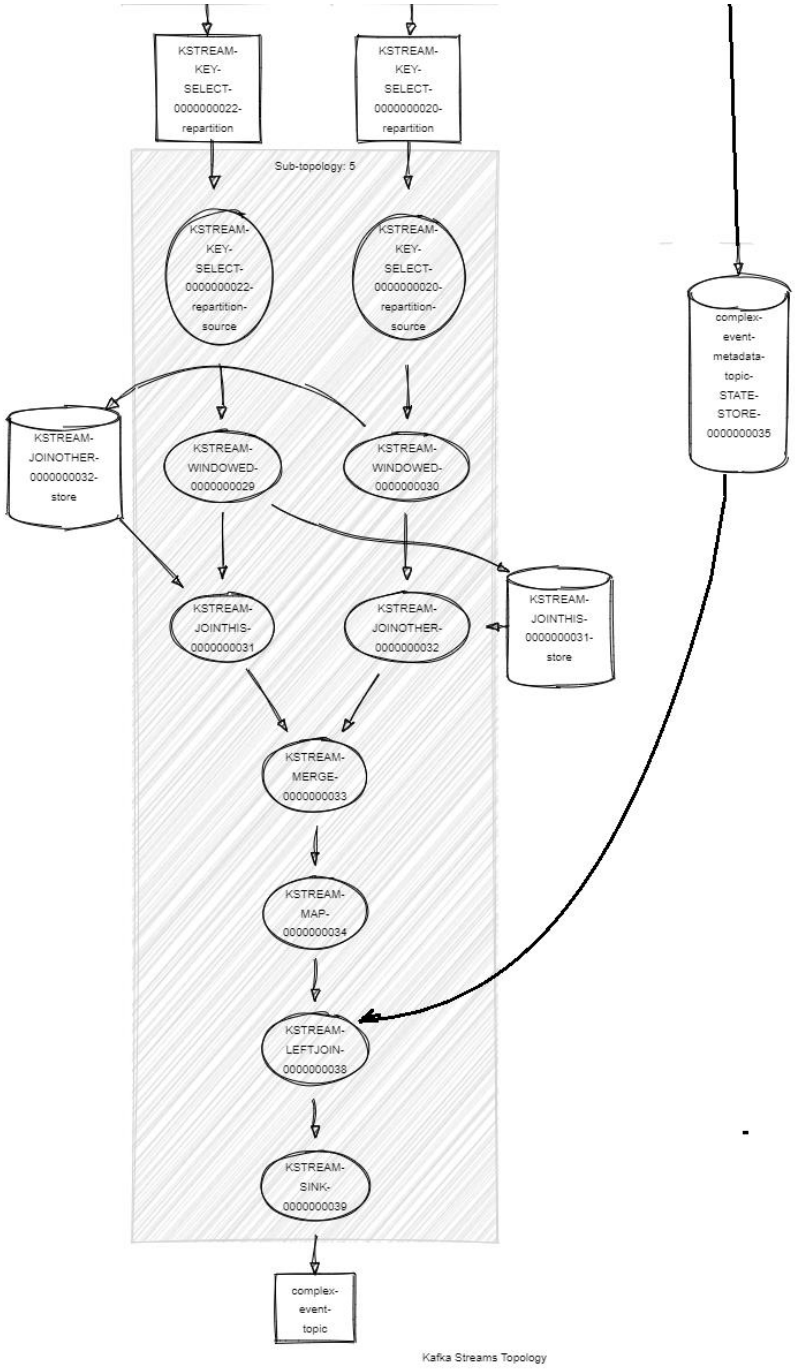

Рисунок 12 – Суб-топология процесса обработки Complex Event (соединения событий Atomic Event и Aggregated Event и последующее обогащение Complex)

Полноценная топология сценария обработки событий представлена в приложении А.

2.3.3 Проектирование компонентной архитектуры

Архитектура слоев, используемая в качестве основного решения, включает в себя следующие основные компоненты(классы):

– Atomic Event Processor, Aggregated Event Processor, Complex Event Processor реализуют функционал по комплексной обработке событий (CEP);

– Atomic Event Mapper, Aggregated Event Mapper, Complex Event Mapper реализуют функционал преобразования входящих событий в доменную бизнес-модель после их обработки классами Processor;

– AtomicEventServer, AggregatedEventServer, ComplexEventServic являются абстракцией для реализации бизнес-модели, что в случае общего сервиса является реализацией механизмов активных правил и служат границей между модулем исполнения активных правил (Data Layer и Business Layer);

– KafkaConfig, KafkaStreamsConfig, KafkaProducerConfig позволяют управлять конфигурациями потребителей и производителей сообщений.

Наглядно архитектура представлена на диаграмме уровня C3 методологии C4 на рисунке 13.

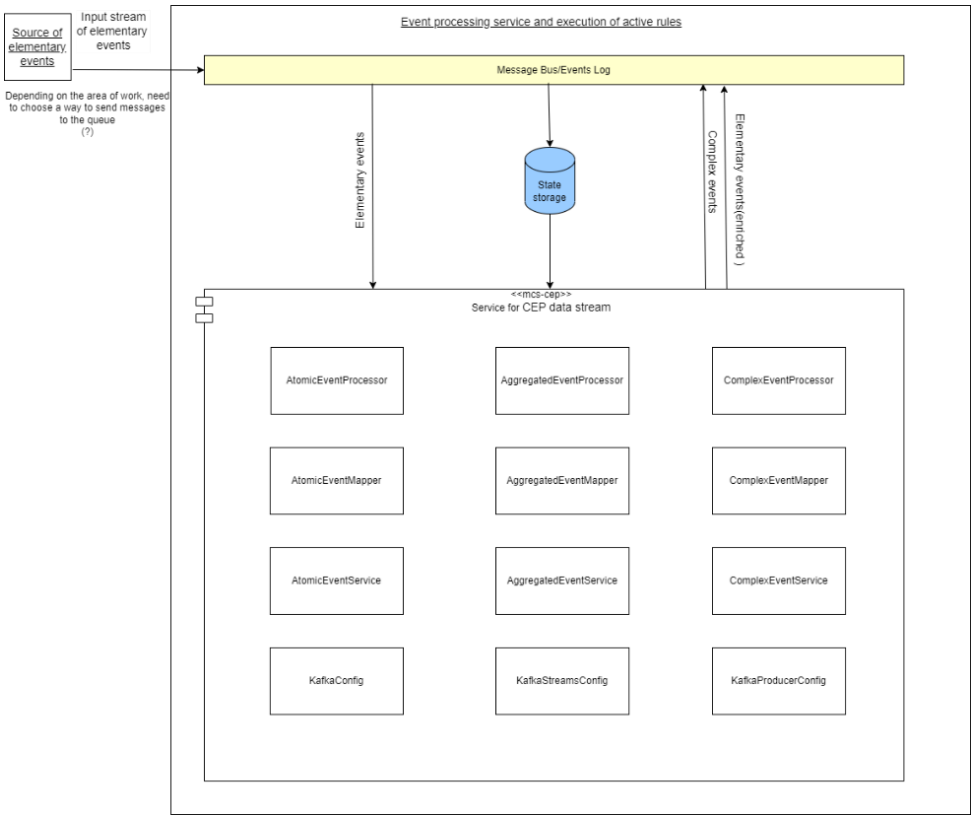

Рисунок 13 – Компонентная архитектура уровня C3 микросервиса потоковой обработки событий

Расширенная схема компонентов представлена в приложении А.

3 Разработка программных средств микросервиса потоковой обработки событий

Данная глава будет представлять конечную реализацию сценария обработки событий под архитектурой микросервиса потоковой обработки событий. Раздел 3.1 будет представлять пример программной реализации CEP, раздел 3.2 будет раскрывать важные настройки распределенной шины, а в разделе 3.3 будет проведено небольшое нагрузочное тестирование, чтобы удостовериться в надежности системы.

3.1 Разработка программных средств потоковой обработки

В предыдущей главе были рассмотрены топологии процессов обработки событий. Реализация для данных топологий представлена на языке Java, используя библиотеку Kafka Streams. Код для графа обработки событий AtomicEvent представлен на листинге 1.

Листинг 1 – Программная реализация топологии обработки событий AtomicEvent

```
@Bean
public KStream<UUID, AtomicEventAvro>
buildAtomicEventsTopology(StreamsBuilder streamsBuilder) {

    var atomicEventsRaw = streamsBuilder.stream(
            atomicEventRawTopic,
            Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventRawSerde())
    );

    var atomicEventsMetadata = streamsBuilder.globalTable(
            atomicEventMetadataTopic,
            Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventMetadataSerde())
    );

    KeyValueMapper<UUID, AtomicEventRawAvro, UUID> keyMapper =
(key, value) ->
            UUID.fromString(value.getAtomicEventMetadataId());
    ValueJoiner<AtomicEventRawAvro, AtomicEventMetadataAvro,
AtomicEventAvro> atomicEventJoiner =
```

```
            atomicEventMapper::toEnriched;
    var atomicEvents = atomicEventsRaw
            .join(atomicEventsMetadata, keyMapper,
atomicEventJoiner);


    atomicEvents.to(atomicEventTopic,
Produced.with(Serdes.UUID(),eventAvroSerdes.createAtomicEventSerde
()));

    return atomicEvents;
}
```

StreamBuilder позволяет строить программную топологию обработки событий. Используя паттерн внедрения зависимостей – dependency injection (DI) фреймворк SpringBoot позволяет без лишних усилий создавать и получать нужные объекты(бины) при помощи аннотаций (пр. @Bean). Также весь жизненный цикл бина полностью контроллируется фреймворком, что позволяет управлять неявным запуском обработчиков потоков событий. Следующая программная реализация топологии обработки Aggregated Event представлена в листинге 2.

Листинг 2 – Программная реализация топологии обработки событий AggrgatedEvent

```
@Bean
public              KStream<UUID,              AggregatedEventAvro>
buildAggregatedEventTopology(StreamsBuilder    streamsBuilder)    {
    var       groupedAtomicEvent       =       atomicEvents.groupBy(
            (key, value) -> value.getHomogeneousCompatibilityId(),
            Grouped.with(Serdes.Integer(),
eventAvroSerdes.createAtomicEventSerde()));
    Initializer<AggregatedEventRawAvro>
aggregatedEventRawInitializer          =          ()          ->
AggregatedEventRawAvro.newBuilder()
            .setId(UUID.randomUUID().toString())
            .setHeterogeneousCompatibilityId(0)
            .setStartTime(Instant.now())
            .setEndTime(Instant.now())
            .setAggregatedEventMetadataId("b4390368-1448-11ee-
```

```
be56-0242ac120002")
            .setAtomicEventIds(new                    LinkedList<>())
            .build();
    Aggregator<Integer,  AtomicEventAvro,  AggregatedEventRawAvro>
aggregatedEventRawAggregator                                        =
            (key,        value,        aggregate)        ->        {
                var   startTimeAtomic   =   value.getStartTime();
                var startTimeAggregated = aggregate.getStartTime();
                var  endTimeAggregated  =  aggregate.getEndTime();
                if        (startTimeAggregated       ==        null
                        ||
startTimeAggregated.isAfter(startTimeAtomic))                       {
                    aggregate.setStartTime(startTimeAtomic);
                }
                if         (endTimeAggregated        ==         null
                        ||
endTimeAggregated.isBefore(startTimeAtomic))                        {
                    aggregate.setEndTime(value.getStartTime());
                }
                var atomicEventIds = aggregate.getAtomicEventIds();
                atomicEventIds.add(value.getId());

aggregate.setHeterogeneousCompatibilityId(value.getHeterogeneousCo
mpatibilityId());
                aggregate.setAggregatedEventMetadataId("b4390368-
1448-11ee-be56-0242ac120002");
                return                                    aggregate;
            };
    var                    aggregatedEventsRaw                    =
            groupedAtomicEvent.aggregate(
                    aggregatedEventRawInitializer,
                    aggregatedEventRawAggregator
            )
                    .toStream()
                    .map((key,               value)              ->
```

```
KeyValue.pair(UUID.fromString(value.getId()),                value));

    var   aggregatedEventsMetadata   =   streamsBuilder.globalTable(
            aggregatedEventMetadataTopic,
            Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventMetadataSerde())
    );

    KeyValueMapper<UUID, AggregatedEventRawAvro, UUID> keyMapper2 =
(key,                              value)                                  ->
            UUID.fromString(value.getAggregatedEventMetadataId());
    ValueJoiner<AggregatedEventRawAvro,
AggregatedEventMetadataAvro,                    AggregatedEventAvro>
atomicEventJoiner2                                                      =
            aggregatedEventMapper::toEnriched;
    var         aggregatedEvents         =         aggregatedEventsRaw
            .join(aggregatedEventsMetadata,              keyMapper2,
atomicEventJoiner2);
    aggregatedEvents.to("aggregated-event-topic",
Produced.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventSerde()));
    return                                          aggregatedEvents;
}
```

Ключевой особенность реализации топологии листинга 2 является написание агрегата событий (Aggregate), который позволяет создавать новый объект и сохранять в нем порцию агрегируемых событий с совместным ключом homogeneousCompatibilityId. Дальнейшая реализация топологии обработки Complex Event повторяет предыдущий код, поэтому в листинге 3 будет представлена только функция слияния событий.

Листинг 3 – Программная реализация топологии обработки событий Complex Event

```
@Bean
public KStream<UUID, ComplexEventAvro>
```

```
buildComplexEventTopology(StreamsBuilder streamsBuilder) {
    var atomicEvents = streamsBuilder.stream(
            atomicEventTopic,
            Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventSerde())
                    //.withTimestampExtractor(new
AtomicEventTimestampExtractor())
    ).selectKey((key,value) ->
value.getHeterogeneousCompatibilityId());

    var aggregatedEvents = streamsBuilder.stream(
            "aggregated-event-topic",
            Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventSerde())
                    //.withTimestampExtractor(new
AggregatedEventTimestampExtractor())
    ).selectKey((key,value) ->
value.getHeterogeneousCompatibilityId());//.map((key, value) ->
KeyValue.pair(value.getHeterogeneousCompatibilityId(), value));

    var joinWindows = JoinWindows.of(
            Duration.ofMinutes(1)).grace(Duration.ofSeconds(10));
    var joinParams = StreamJoined
            .with(
                    Serdes.Integer(),
                    eventAvroSerdes.createAggregatedEventSerde(),
                    eventAvroSerdes.createAtomicEventSerde()
            );
    ValueJoiner<AggregatedEventAvro, AtomicEventAvro,
ComplexEventRawAvro> valueJoiner =
            (aggregatedEventAvro, atomicEventAvro) -> {
                var complexEventRawAvro =
ComplexEventRawAvro.newBuilder()
                        .setId(UUID.randomUUID().toString())
                        .setStartTime(Instant.now())
                        .setEndTime(Instant.now())
                        .setAtomicEventId(atomicEventAvro.getId())

.setAggregatedEventId(aggregatedEventAvro.getId())
                        .setComplexEventMetadataId("8cfeab54-1448-
11ee-be56-0242ac120002")
                        .build();
                var atomicEventStartTime =
atomicEventAvro.getStartTime();
                var aggregatedEventStartTime =
aggregatedEventAvro.getStartTime();
                complexEventRawAvro.setStartTime(

aggregatedEventStartTime.isBefore(atomicEventStartTime)
                                ? aggregatedEventStartTime :
atomicEventStartTime
                );
```

```
                var aggregatedEventEndTime =
aggregatedEventAvro.getStartTime();
                complexEventRawAvro.setStartTime(

aggregatedEventEndTime.isAfter(atomicEventStartTime)
                                    ? aggregatedEventEndTime :
atomicEventStartTime
                );
                return complexEventRawAvro;
            };
    var complexEventsRaw = aggregatedEvents
            .join(atomicEvents, valueJoiner, joinWindows,
joinParams)
            .map((key, value) ->
KeyValue.pair(UUID.fromString(value.getId()), value));

    var complexEventsMetadata = streamsBuilder.globalTable(
            complexEventMetadataTopic,
            Consumed.with(Serdes.UUID(),
eventAvroSerdes.createComplexEventMetadataSerde())
    );
    KeyValueMapper<UUID, ComplexEventRawAvro, UUID> keyMapper3 =
(key, value) ->
            UUID.fromString(value.getComplexEventMetadataId());
    ValueJoiner<ComplexEventRawAvro, ComplexEventMetadataAvro,
ComplexEventAvro> complexEventJoiner =
            complexEventMapper::toEnriched;
    var complexEvent = complexEventsRaw
            .join(complexEventsMetadata, keyMapper3,
complexEventJoiner);

    complexEvent.to("complex-event-topic",
Produced.with(Serdes.UUID(),
eventAvroSerdes.createComplexEventSerde()));
    return complexEvent;
}
```

Важным моментом слияния событий Atomic Event и Aggregated Event является их контекст потока (KStream). Предыдущие слияния проводились с табличной абстракцией (KTable), когда один из потоков накапливал события в хранилище состояний, тем самым реализовывал поиск идентичный поиску по БД. Слияние простых потоков не может предоставить такой функционал, взамен этого предоставляются оконные функции ожидания. Окна позволяют группировать события в привязке к явным интервалам времени, тем самым осуществляя операции агрегации/слияния и тому подобными.

Полноценная кодовая база представлена в разделе приложения Г.

3.2 Реализация программных средств распределенной шины сообщений

Главная шина сообщений в виде Apache Kafka реализовывалась с помощью функциональности Docker [43]. Docker позволяет «упаковывать» приложение со всем его окружением и зависимостями в контейнер, с целью дальнейшего использования. Топология брокеров представлена на рисунке 16.

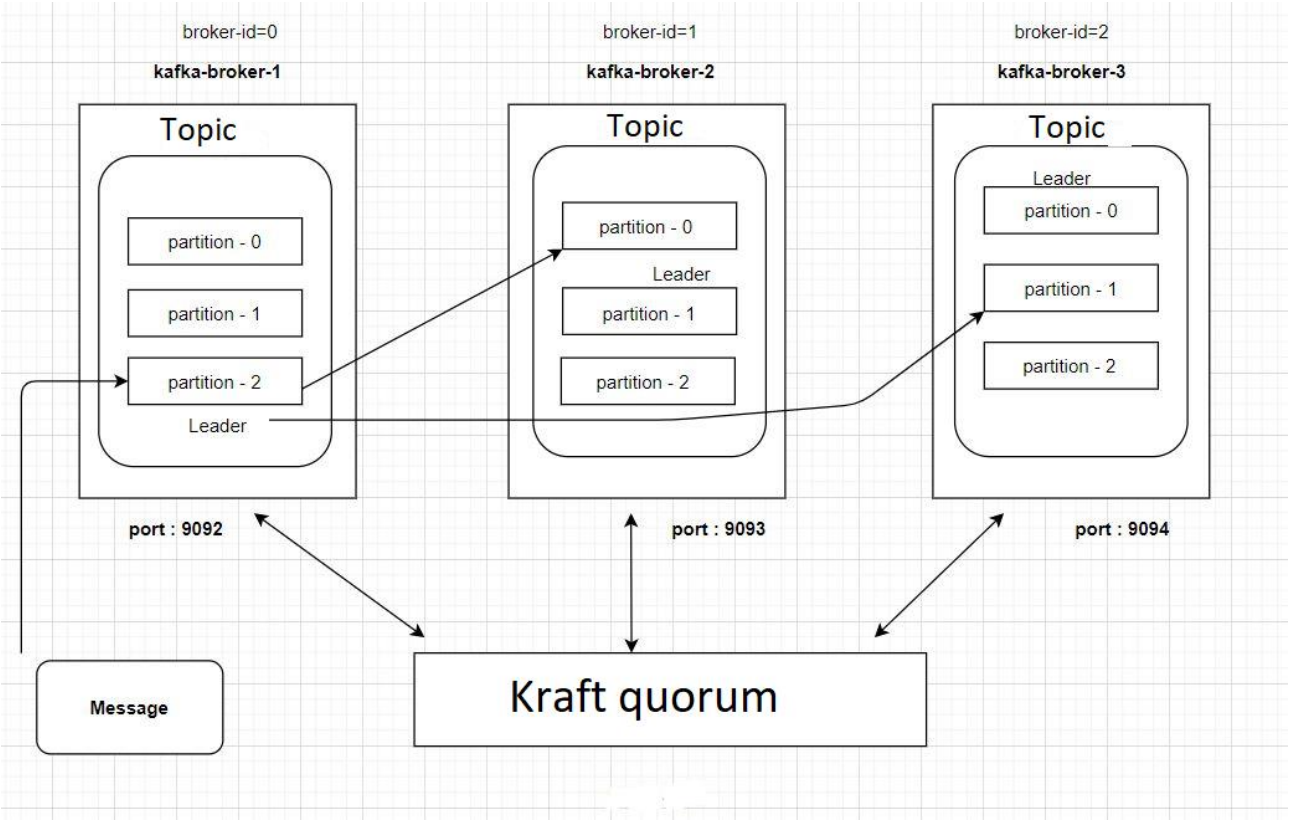

Рисунок 14 – Топология брокеров микросервиса потоковой обработки событий

Топология Kafka представлена тремя брокерам на кластер и четырьмя разделами на тему. Данная решение было принято в связи с обеспечением минимальной отказоустойчивости/надежности хранилища событий и минимально требуемой пропускной способности. Данные качества проявляются через репликацию разделов на ведомые узлы(master-slave). Весь кластер управляется технологией Kraft реализующейся через собрание(кворум) брокеров. Кворум создан для выборки главного брокера – лидера, который и

будет открывать входное соединение с целью получения сообщений, а потом их репликации на остальные узлы.

Также функцией отказоустойчивости брокера является инициация лидером кластера перебалансировки разделов, которые изначально распределяются по кластеру. Данная процедура призвана снизить нагрузку на брокеров, количество разделов на которых дает большой процент перегрузки.

Еще одним важным механизмом Apache Kafka является возможность масштабирования. Это достигается за счет увеличения количества разделов на одну тему. Раздел является атомарной единицей брокера, хранящей сообщения. Фактически раздел – это и есть очередь сообщений. Среднее рекомендуемое число раздело на тему – четыре.

В функциональных требованиях главы 2 были указаны возможности настройки/конфигурации распределенной шины сообщений на предмет обговоренной функциональности (надежности, отказоустойчивости, масштабирования). Данная функциональность реализуется по средствам настройки внешних(environment) переменных брокера в кластере, которые затрагивают внутренний файл конфигурации каждого узла. В данной работе практически все параметры брокера будут оставлены в своих стандартных показателях

Далее представлены параметры отказоустойчивости и надежности.

broker-heartbeat-interval-ms/broker-sessio-ms/replica-lag-time-max-ms/controller-quorum-fetch-timeout-ms/controller-quorum-election-timeout. Данные настройки являются основными тайм-аутами брокера, позволяя заранее выявлять неактивных участников кластера и отключать их или выявлять отстающих в репликации узлов (несинхронизированных реплик), с дальнейшей целью перебалансировки и сохранения консистентности. Также влияют на параметр латентности.

auto-leader-rebalance-enable. Основная настройка включения автоматического отслеживания и перебалансировки разделов.

auto-create-topic-senable/num-partition/max-connections/background-threads. Основные настройки масштабирования пропускной способности брокера.

Для более детальной настройки брокера рекомендуется обратиться к документации. Полная настройка брокера сообщений представлена в приложении В.

Также с Apache Kafka тесно связаны еще две сущности – производители сообщений и их потребители. Сущности реализуются на стороне приложения, работающего с брокерами. Производителей и потребителей также можно конфигурировать. Данная возможность была реализована при помощи чтения файла из «коробки» фреймворком SpringBoot и его паттернов DI/IoC.

У производителя сообщений, как и у брокера, есть настройки таймаутов отправки сообщения linger-ms/request-timeout-ms/retry-backoff-ms/delivery-timeout-ms, но более важной настройкой является подтверждение гарантированности и консистентности доставки. За данные настройки отвечают параметры acks, enable-idempotence, max-in-flight-requests-per-connection и partitioner.

Потребитель сообщений конфигурируется похожим образом, за исключением двух параметров enable-auto-commit и partition-assignment-strategy. Первый параметр отвечает за сохранение позиции смещения чтения – место в очереди, с которого начинается читка сообщений, второй же определяет способ назначения потребителя на чтение конкретного раздела.

Для более детальной настройки производителей и потребителей рекомендуется обратиться к документации. Полная настройка данных сущностей представлена в приложении В.

Аналогично Kafka Streams предоставляет возможность конфигурации. Так как библиотека создана на основе Apache Kafka, то внутри реализации скрываются все те же потребители и производители, конфигурирующиеся описанным ранее способом. Важными настройками, которые позволяет изменять библиотека являются параметр масштабируемости num-stream-threads,

и параметры репликации: replication-factor/num-standby-replicas/max-warmup-replicas. Первый параметр отвечает за количество копий одного и того же потребителя, так как в зависимости от этого параметра запускается указанное количество потоков, в каждом из которых начинает работать копия созданного потребителя, и это правило распространяется на все созданные потребители. Остальные три параметра отвечают за репликацию хранилищ состояний, использующихся в потоковой обработке.

Для более детальной настройки Kafka Streams рекомендуется обратиться к документации. Полная настройка представлена в приложении Б.

3.3 Тестирование программных средств

В данном разделе представлено тестирование работы реализованного сценария обработки событий, а также результаты нагрузочного тестирования, произведенного при помощи встроенных утилит Kafka. Все тесты проводились на конфигурации, указанной в предыдущем разделе. Для проверки функциональности использовались утилиты Postman, для посылки сообщения в Kafka и Offset Explorer, для проверки работоспособности кластера.

Функциональное тестирование сценария состояло из действия по посылке сообщения в кластер Kafka, который переправлял его потребителю – микросервису потоковой обработки событий. Микросервис принимал сообщение и на основании сценария задействовал механизмы CEP. Наглядные примеры тест-кейсов находятся в приложении В.

Нагрузочное тестирование представляет из себя подачу большого количества данных на обработку в кластер приложения с целью получения метрик. Стоит учитывать, что результаты нагрузочного тестирования являются лишь статическим показателем пропускной способности и следует тестировать нагрузку непосредственно в развертываемой среде, также тестирование кластера зависит от машины, на которой система была развернута. В данном случае характеристики представлены в таблице 1.

Таблица 1 – Характеристики аппаратно-технического обеспечения

Характеристики	Минимальное возможное значение	Максимальное возможное значение
Процессор	Тактовая частота 2,9 ГГц	4,2 ГГЦ
Количество вычислительных потоков процессора	4/8	4/8
Оперативная память	16 ГБ/4 Гб (Vimm)	16 ГБ/16 Гб (Vimm)
Свободное дисковое пространство	15 ГБ	15 ГБ
Тип дисковых накопителей	SSD /NVMe	SSD /NVMe

Сценарий нагрузочного тестирования состоял из трех этапов: посылка 1000/10000/100000 записей. Посылка осуществлялась с помощью вызова утилиты Kafka, сигнатура которой представлена в листинге 4.

Листинг 4 – Вызов утилиты тестирования на запись

```
kafka-producer-perf-test --topic atomic-event-raw-topic --num-
records 1000 --throughput 100 --producer-props
bootstrap.servers=broker-first:29092,broker-second:29093,broker-
third:29094
key.serializer=org.apache.kafka.common.serialization.StringSeriali
zer
value.serializer=org.apache.kafka.common.serialization.StringSeria
lizer --record-size 204800 --print-metrics
```

Основные полученные метрики тестирования представлены на рисунках 17 - 19. Полный набор метрик представлен в приложении В.

Рисунок 15 – Метрики тестирования кластера на запись 1000 сообщений

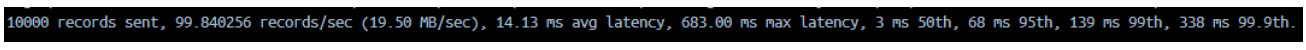

Рисунок 16 – Метрики тестирования кластера на запись 10000 сообщений

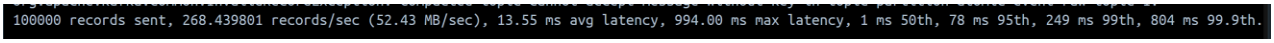

Рисунок 17 – Метрики тестирования кластера на запись 10000 сообщений

Цифры, полученные с метрик нагрузочного тестирования, являются стандартными [44] для брокера без специфической настройки и отражают явную способность к дальнейшему росту.

Заключение

В заключение данной работы было успешно достигнуто несколько целей, связанных с разработкой архитектурного и технологического решения для микросервиса потоковой обработки событий.

В первую очередь, была организована способность оперировать потоком гетерогенных событий путем непрерывных вычислений без необходимости явного сбора или сохранения каждого отдельного события. Это обеспечило быстрое и эффективное выявление сложных событий, основанное на модели CEP, во входящем потоке данных.

Наконец, в процессе разработки была обеспечена надежность и масштабируемость шины данных, а также самого микросервиса потоковой обработки событий. Это позволяет обрабатывать большие объемы данных с высокой производительностью и гарантирует устойчивую работу системы даже при возникновении нагрузок или сбоев.

В итоге, созданное архитектурное и технологическое решение является надежным, масштабируемым и эффективным инструментом для обработки потоковых событий. Оно отвечает поставленным целям и может быть успешно применено в различных сценариях, требующих обработки и анализа данных в реальном времени.

Список использованных источников

1. Hadoop, MapReduce Overview // [Электронный ресурс]. URL.: https://hadoop.apache.org/docs/stable1/mapred_tutorial.html (дата обращения:13.04.2023).

2. Пселтис Эндрю Дж, Потоковая обработка. - СПб.: ДМК-Пресс, 2018.

3. Di Wang Worcester Polytechnic Institute Worcester, MA, USADi Wang, Elke A. Rundensteiner, Richard T. Ellison III. Active Complex Event Processing over Event Streams // ResearchGate [Электронный ресурс]. URL: https://www.researchgate.net/publication/220538624_Active_Complex_Event_Processing_over_Event_Streams (дата обращения: 13.04.2023).

4. Dong Wang, Mingquan Zhou, Sajid Ali, Pengbo Zhou, Yusong Liu, Xuesong Wang, A Novel Complex Event Processing Engine for Intelligent Data Analysis in Integrated Information Systems // ResearchGate [Электронный ресурс]. URL:https://www.researchgate.net/publication/298797020_A_Novel_Complex_Event_Processing_Engine_for_Intelligent_Data_Analysis_in_Integrated_Information_Systems (дата обращения: 13.04.2023).

5. 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: 13.04.2023).

6. Martin Kleppman, Turning the database inside out with Apache Samza // Strange Loop Conference [Электронный ресурс]. URL: https://www.youtube.com/watch?v=fU9hR3kiOK0 (дата обращения: 15.04.2023).

7. Ben Stopford, Messaging as the Single Source of Truth // Confluent [Электронный ресурс]. URL: https://www.confluent.io/blog/messaging-single-source-truth/ (дата обращения: 15.04.2023).

8. Ben Stopford, Event Streams as a Shared Source of Truth // Oreily [Электронный ресурс]. URL: https://www.oreilly.com/library/view/designing-event-driven-systems/9781492038252/ch09.html (дата обращения: 15.04.2023).

9. 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: 15.04.2023).

10. Dr. Gopala, Raju Myadam, Enterprise Reference Architecture: A Primer on IoT-Based Systems // Medium [Электронный ресурс]. URL: https://medium.com/@wiprodigital/enterprise-reference-architecture-a-primer-on-iot-based-systems-2c243953ac33 (дата обращения: 15.04.2023).

11. Ali N. M., Big Data: Analytical Solutions, Research Challenges and Trends // Публикации ВШЭ [Электронный ресурс]. URL: https://publications.hse.ru/articles/352038681 (дата обращения: 15.04.2023).

12. Буч Г., Язык UML, Руководство пользователя. 2-е изд.: Пер. с англ. Мухин Н. – М.: ДМК Пресс, 2006. – 496 с.: ил.

13. 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.

14. Шибанов С. В., Метамодель сервиса потоковой обработки событий / С. В. Шибанов, Я. С. Шлепнев, А. С. Гусаров // Методы, средства и технологии получения и обработки измерительной информации («Шляндинские чтения - 2022») : материалы XIV Международной научно-технической конференции с элементами научной школы и конкурсом научно-исследовательских работ для

обучающихся и молодых ученых, Пенза, 24–26 октября 2022 года. – Пенза: Пензенский государственный университет, 2022. – С. 240-243. – EDN RUVERT.

15. Dr. Martin Kleppmann, Distributed Systems, University of Cambridge Computer Science Tripos, Part IB Michaelmas term 2021/22.

16. Chris Richardson, What are microservices? // Microservices [Электронный ресурс]. URL: https://microservices.io/index.html (дата обращения:1.05.2023).

17. Крис Ричардсон, Микросервисы, Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. – 544 с.: ил. — (Серия «Библиотека программиста»).

18. Ньюмен С, Создание микросервисов. СПб.: Питер, 2016. — 304 с.: ил. - (Серия «Бестселлеры O’Reilly»).

19. 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).

20. 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.

21. Martin Fowler, Focusing on Events // MartinFowler [Электронный ресурс]. URL: https://martinfowler.com/eaaDev/EventNarrative.html (дата обращения:1.05.2023).

22. C4 model. The C4 model for visualising software architecture // C4 [Электронный ресурс]. URL: https://c4model.com/ (дата обращения:1.05.2023)

23. 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.

24. Mike Ralphson, OpenAPI Specification // GitHub [Электронный ресурс]. URL: https://github.com/OAI/OpenAPI-Specification (дата обращения: 2.05.2023).

25. BigData School, CAP // BigDataSchool [Электронный ресурс]. URL: https://bigdataschool.ru/wiki/cap (дата обращения: 2.05.2023).

26. Stafford P, What is Mifid II and how will it affect EU’s financial industry? Financial Times. September 15, 2017. – URL: https://www.ft.com/content/ae935520-96ff-11e7-b83c-9588e51488a0 (дата обращения: 2.05.2023).

27. Роберт Мартин, Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2018. – 352 с.: ил. (Серия «Библиотека программиста»).

28. Matthias Noback. Layers, Ports, Adapters P2// matthiasnoback.nl. URL: https://matthiasnoback.nl/2017/08/layers-ports-and-adapters-part-2-layers/ (дата обращения: 2.05.2023).

29. Matthias Noback. Layers, Ports, Adapters P3 // matthiasnoback.nl. URL: https://matthiasnoback.nl/2017/08/layers-ports-and-adapters-part-3-ports-and-adapters/ (дата обращения: 2.05.2023).

30. Apache Kafka, Kafka Intro // kafka.apache.org URL: https://kafka.apache.org/intro/ (дата обращения: 5.05.2023).

31. Apache Pulsar, Pulsar Docs // pulsar.apache.org URL: https://pulsar.apache.org/docs/3.0.x/ (дата обращения: 5.05.2023)

32. RabbitMQ, RabbitMQ Docs // pulsar.apache.org URL: https://www.rabbitmq.com/ (дата обращения: 5.05.2023).

33. Alok Nikhil, Vinoth Chair, Benchmarking Apache Kafka, Apache Pulsar, and RabbitMQ: Which is the Fastest? // Comfluent URL: https://www.confluent.io/blog/kafka-fastest-messaging-system/#throughput-test (дата обращения: 5.05.2023).

34. Apache Storm, Storm Docs // storm.apache.org URL: https://storm.apache.org/releases/2.4.0/index.html (дата обращения: 5.05.2023).

35. Apache Spark, Spark Docs // spark.apache.org URL: https://spark.apache.org/docs/latest/ (дата обращения: 5.05.2023).

36. Apache Flink, Flink Docs // flink.apache.org URL: https://flink.apache.org/ (дата обращения: 5.05.2023).

37. Kafka Streams, Spark Docs // kafka.apache.org URL: https://kafka.apache.org/documentation/streams/ (дата обращения: 5.05.2023).

38. Apache Samza, Samza Docs // samza.apache.org URL: https://samza.apache.org/learn/documentation/latest/core-concepts/core-concepts.html (дата обращения: 5.05.2023).

39. https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b,

40. Самарев Р.С., Обзор состояния области потоковой обработки данных. Труды ИСП РАН, том 29, вып. 1, 2017 г., стр. 231-260. DOI: 10.15514/ISPRAS2017-29(1)-13.

41. Спилкэ Лауренцу, Spring быстро. – СПб.: Питер, 2023. – 448 с.: ил. ISBN 978-5-4461-1969-1.

42. Tiago Boldt Sousa, Dataflow Programming: Concept, Languages and Applications // Doctoral Symposium on Informatics Engineering URL:https://www.researchgate.net/publication/281777036_Dataflow_Programming_Concept_Languages_and_Applications (дата обращения: 5.05.2023).

43. Docker Container, Docker Docs // docker.com URL: https://www.docker.com/ (дата обращения: 7.05.2023)

44. LinkedIn, Benchmarking Kafka // linkedIn.com URL: https://engineering.linkedin.com/kafka/benchmarking-apache-kafka-2-million-writes-second-three-cheap-machines

Приложение А

АРХИТЕКТУРА

(рекомендуемое)

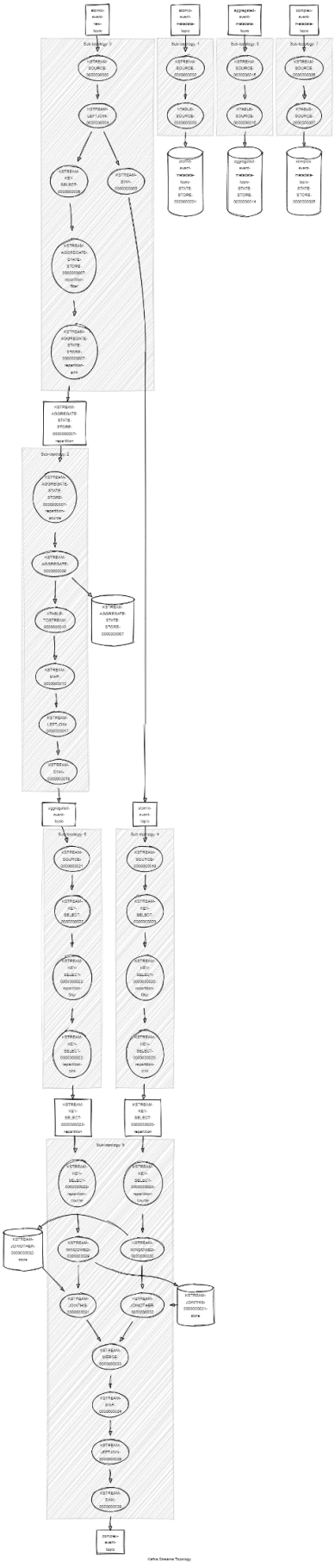

Рисунок А.1 – Полная топология процесса сценария обработки событий

Рисунок А.2 – Полная схема компонентов микросервиса потоковой обработки событий

Приложение Б

КОНФИГУРАЦИЯ СИСТЕМЫ

(рекомендуемое)

Листинг Б.1 – YAML файл конфигурации потребителей, производителей и KafkaStreams.

```
server:
  port: ${APP_PORT:8080}

spring:
  application:
    name: "mcs-complex-event-processing"

  kafka:
    bootstrap-servers:
"localhost:9092,localhost:9093,localhost:9094"

    producer:
      key-serializer:
org.apache.kafka.common.serialization.UUIDSerializer #(?)
      value-serializer:
io.confluent.kafka.serializers.KafkaAvroSerializer #(?)
      properties:
        # Настройка надежности
        ## Отказоустойчивость/доступность
        ### Настройки лимитов отправляемых данных
        batch-size: 204800 #(?)
        max-request-size: 204800 #(?)
        ### Настройка лимитов буфера ожидания
        buffer-memory: 33554432
        ### Настройки тайм-аутов блокировки буфера ожидания
        max-block-ms: 60000
        ### Настройки тайм-аутов доставки
        linger-ms: 20 #(?)
        request-timeout-ms: 30000
        retry-backoff-ms: 100
        delivery-timeout-ms: 120000
        ### Настройка повторных попыток
        retries: 2147483647
        ### Настройки гарантий доставки
        acks: "all" #(?)

        ## Консистентность доставки
        enable-idempotence: true #(?)

        # Настройка масштабируемости
        max-in-flight-requests-per-connection: 3 #(?)

    streams:
      client-id: "atomic-event"
      application-id: "atomic-event-stream"
      properties:
        #! Настройка задач
        # Настройка получения сообщений
```

```
        default-key-serde:
org.apache.kafka.common.serialization.Serdes.UUIDSerde #(?)
        default-value-serde:
io.confluent.kafka.streams.serdes.avro.SpecificAvroSerde #(?)

        # Настройка надежности
        ## Отказоустойчивость/доступность
        ### Настройки тайм-аутов потребителя задачи
        task-timeout-ms: 300000
        commit-interval-ms: 100 #depend from processing-guarantee
        ### Настройки тайм-аутов брокера
        metadata-max-age-ms: 300000
        ### Оптимизация данных на пути топологии
        buffered-records-per-partition: 1000
        cache-max-bytes-buffering: 10485760
        repartition-purge-interval-ms: 30000
        topology-optimization: "all" #(?)
        ### Exception-handlers
        default-deserialization-exception-handler:
org.apache.kafka.streams.errors.LogAndFailExceptionHandler
        default-production-exception-handler:
org.apache.kafka.streams.errors.DefaultProductionExceptionHandler

        ## Настройки консистентности обработки
        max-task-idle-ms: 0
        processing-guarantee: "exactly_once_v2" #(?)

        # Настройка масштабируемости
        num-stream-threads: 2 #(?)

        #! Настройки состояний
        # Настройки хранения состояний
        default-dsl-store: "rocksDB"
        #state-dir: "/tmp/kafka-streams"
        statestore-cache-max-bytes: 10485760
        # Настройки внешнего доступа к состояниям
        application-server: ''
        # Конфигурация RocksDB
        rocksdb-config-setter: null

        # Настройка надежности
        ## Отказоустойчивость/доступность
        ### Настройка лимитов
        acceptable-recovery-lag: 10000
        ### Тайм-ауты перебалансировки
        probing-rebalance-interval-ms: 600000
        state-cleanup-delay-ms: 600000
        ### Репликация
        replication-factor: -1
        num-standby-replicas: 1 #(?) !
        max-warmup-replicas: 1 #(?) !
```

```
        # Внутренний потребитель под kstreams
        consumer:
          # Идентификация потребителя
          group-id: null
          client-id: null
          client-rack: null
          #group-instance-id: "atomic-event-consumer-instance"
#(?)

          # Настройка надежности
          ## Отказоустойчивость/доступность
          ### Настройки лимитов читаемых данных
          fetch-min-bytes: 1
          fetch-max-bytes: 52428800
          max-poll-records: 1000 #default for streams
          max-partition-fetch-bytes: 1048576
          ### Настройки тайм-аутов потребителя
          fetch-max-wait-ms: 500
          heartbeat-interval-ms: 3000
          session-timeout-ms: 45000
          max-poll-interval-ms: 300000
          auto-commit-interval-ms: 100
          ### Настройки тайм-аутов брокера
          request-timeout-ms: 30000
          retry-backoff-ms: 100
          default-api-timeout-ms: 60000
          reconnect-backoff-ms: 50
          reconnect-backoff-max-ms: 1000
          ### Настройки гарантий чтения
          auto-offset-reset: "earliest" #default for streams
          enable-auto-commit: false #default for streams
          ### Стратегия назначения/балансировка партиций
          partition-assignment-strategy:
org.apache.kafka.clients.consumer.CooperativeStickyAssignor #(?)

        main-consumer: #(?)
        restore-consumer: #(?)
        global-consumer: #(?)
        producer: #(?)

        # Метрики
        metrics-recording-level: "INFO"

        # Конфигурация перехода на обновление
        upgrade-from: null

    schema:
      schema-registry-url: "http://localhost:8081/"
      key-subject-name-strategy:
io.confluent.kafka.serializers.subject.RecordNameStrategy
      value-subject-name-strategy:
io.confluent.kafka.serializers.subject.RecordNameStrategy
```

```
      auto-register-schemas: true
      use-latest-version: false
      avro-deserializer-config:
        specific-avro-reader: true

    topics:
      atomic-raw: "atomic-event-raw-topic"
      aggregated-raw: "aggregated-event-raw-topic"
      complex-raw: "complex-event-raw-topic"

      atomic: "atomic-event-topic"
      aggregated: "aggregated-event-topic"
      complex: "complex-event-topic"

      atomic-metadata: "atomic-event-metadata-topic"
      aggregated-metadata: "aggregated-event-metadata-topic"
      complex-metadata: "complex-event-metadata-topic"

  boot:
    admin:
      client:
        url: "http://localhost:5005/admin"
      context-path: "/admin"

management:
  endpoints:
    web:
      exposure:
        include: "*"

logging:
  file:
    path: "log"
  level:
    ROOT: info
    org.springframework.security: trace
    org.hibernate: info
  logback:
    rollingpolicy:
      clean-history-on-start: true
```

Листинг Б.2 – YAML файл конфигурации брокера шины сообщений

```
broker:
  image: confluentinc/cp-kafka:latest
  hostname: "broker-first"
  container_name: "broker-first"
  ports:
    - "9092:9092"
    - "9101:9101"
  environment:
    # Базовые настройки идентификации
```

```
    KAFKA_NODE_ID: 1
    CLUSTER_ID: "MkU3OEVBNTcwNTJENDM2Qk"
    #KAFKA_BROCKER_RACK: "rack"
    ## Настройки брокера как контроллера KRaft
    KAFKA_PROCESS_ROLES: "broker,controller"
    KAFKA_CONTROLLER_QUORUM_VOTERS: "1@broker-
first:29093,2@broker-second:29094,3@broker-third:29095"
    #CONFLUENT_SUPPORT_CUSTOMER_ID: "anonymous"

    # Настройка точек доступа брокера
    ### Имя прослушивателя запросов контроллера
    KAFKA_CONTROLLER_LISTENER_NAMES: "CONTROLLER"
    ### Имя прослушивателя запросов брокера
    KAFKA_INTER_BROKER_LISTENER_NAME: "BROKER"
    ### Точка доступа для подключения к брокеру/контроллеру(из
внешней/внутренней сети docker)
    ### [имя прослушивателя:имя_хоста_внутри_docker:порт]
    KAFKA_LISTENERS: "BROKER_HOST://broker-
first:9092,BROKER://broker-first:29092,CONTROLLER://broker-
first:29093"
    ### Точка доступа по получению метаданных
брокера/контроллера(из внешней/внутренней сети docker) - его точек
подключения продьюсера/консьюмера
    KAFKA_ADVERTISED_LISTENERS:
"BROKER_HOST://localhost:9092,BROKER://broker-first:29092"
    ### ключ/значение для используемого протокола безопасности в
зависимости от имени прослушивателя
    KAFKA_LISTENER_SECURITY_PROTOCOL_MAP:
"CONTROLLER:PLAINTEXT,BROKER:PLAINTEXT,BROKER_HOST:PLAINTEXT"

    # Настройка параметров надежности брокера
    ## Отказоустойчивость/надежность
    ### Настройка тайм-аутов брокера
    KAFKA_INITIAL_BROKER_REGISTRATION_TIMEOUT_MS: 60000
    KAFKA_BROKER_HEARTBEAT_INTERVAL_MS: 200
    KAFKA_BROKER_SESSION_TIMEOUT_MS: 60000
    KAFKA_REPLICA_LAG_TIME_MAX_MS: 1000
    KAKFA_REPLICA_FETCH_BACKOFF_MS: 1000 # параметр не существует
    KAKFA_REPLICA_FETCH_WAIT_MAX_MS: 500 # параметр не существует
    KAFKA_REPLICA_SOCKET_TIMEOUT_MS: 30000
    ### Настройка тайм-аутов группы контроллеров
    KAFKA_CONTROLLER_QUORUM_FETCH_TIMEOUT_MS: 2000
    KAFKA_CONTROLLER_QUORUM_ELECTION_TIMEOUT_MS: 1000
    KAFKA_CONTROLLER_QUORUM_ELECTION_BACKOFF_MAX_MS: 1000
    KAFKA_CONTROLLER_QUORUM_RETRY_BACKOFF_MS: 20
    ## Реплицируемость
    #KAFKA_DEFAULT_REPLICATION_FACTOR: 3 #(?)
    #KAFKA_MIN_INSYNC_REPLICAS: 2 #(?)
    ### Настройка политик перебалансировки разделов
    KAFKA_AUTO_LEADER_REBALANCE_ENABLE: true
    KAFKA_LEADER_IMBALANCE_PER_BROKER_PERCENTAGE: 10
    KAFKA_LEADER_IMBALANCE_CHECK_INTERVAL_SECONDS: 300
```

```
# Настройка параметров консистентности брокера
## Политика выбора лидера с консистентными данными
#KAFKA_UNCLEAN_LEADER_ELECTION_ENABLE: false

# Настройка параметров масштабирования
### Настройка масштабирования разделов
KAFKA_AUTO_CREATE_TOPICS_ENABLE: true
KAFKA_DELETE_TOPIC_ENABLE: true
KAFKA_NUM_PARTITIONS: 4 #(?)
### Настройка очереди сообщений
KAFKA_MESSAAGE_MAX_BYTES: 1048576
KAFKA_QUEUE_MAX_REQUESTS: 500
KAFKA_QUEUED_MAX_REQUEST_BYTES: -1
### Настройка подключений
KAFKA_MAX_CONNECTIONS: 2147483647
KAFKA_CONNECTIONS_MAX_IDLE_MS: 600000
### Настройка потоков обработки
KAFKA_BACKGROUND_THREADS: 10
KAFKA_NUM_IO_THREADS: 8
KAFKA_NUM_NETWORK_THREADS: 3
KAFKA_NUM_RECOVERY_THREADS_PER_DATA_DIR: 1
KAFKA_NUM_REPLICA_FETCHERS: 3

# Настройка тайм-аутов для группы потребителей
KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 3000
KAFKA_GROUP_MIN_SESSION_TIMEOUT_MS: 6000
KAFKA_GROUP_MAX_SESSION_TIMEOUT_MS: 1800000

# Настройка хранения/отказоустойчивости смещений
KAFKA_OFFSETS_COMMIT_TIMEOUT_MS: 5000
KAFKA_OFFSETS_RETENTION_CHECK_INTERVAL_MS: 600000
KAFKA_OFFSETS_COMMIT_REQUIRED_ACKS: -1
KAFKA_OFFSETS_LOAD_BUFFER_SIZE: 5242880
KAFKA_OFFSETS_RETENTION_MINUTES: 10080
KAFKA_OFFSETS_TOPIC_NUM_PARTITIONS: 50
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 3 #?
KAFKA_OFFSETS_TOPIC_SEGMENT_BYTES: 104857600

# Настройка транзакций
#KAFKA_CONFLUENT_LICENSE_TOPIC_REPLICATION_FACTOR: 1
#KAFKA_CONFLUENT_BALANCER_TOPIC_REPLICATION_FACTOR: 1
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1 #?
KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1 #?

# Настройка политик хранения сообщений
#KAFKA_LOG_DIR:
#KAFKA_METADATA_LOG_DIR:
KAFKA_LOG_RETENTION_HOURS: 168
KAFKA_LOG_RETENTION_BYTES: -1
KAFKA_LOG_SEGMENT_BYTES: 1073741824
KAFKA_LOG_ROLL_HOURS: 168
```

```
    KAFKA_FLUSH_INTERVAL_MESSAGES: 9223372036854775807
    KAFKA_FLUSH_INTERVAL_MS: 9223372036854775807

    KAFKA_LOG_CLEANER_ENABLE: true
    KAFKA_LOG_CLEANUP_POLICY: "compact" #(?)
    KAFKA_LOG_CLEANER_MIN_CLEANABLE_RATIO: 0.5
    KAFKA_LOG_CLEANER_MIN_COMPACTION_LAG_MS: 1
    KAFKA_LOG_CLEANER_MAX_COMPACTION_LAG_MS: 9223372036854775807
    KAFKA_LOG_RETENTION_CHECK_INTERVAL_MS: 300000

    # Настройка метрик
    KAFKA_METRIC_REPORTERS: ''
    KAFKA_METRICS_POLLING_INTERVAL_SECS: 10
    ## JMX
    KAFKA_JMX_PORT: 9101 #(?)
    KAFKA_JMX_HOSTNAME: "localhost" #(?)

  volumes:
    - ./brokers/broker-first/log:/var/lib/kafka/log
    - ./brokers/broker-first/data:/var/lib/kafka/data
```

Приложение В

ТЕСТИРОВАНИЕ

(рекомендуемое)

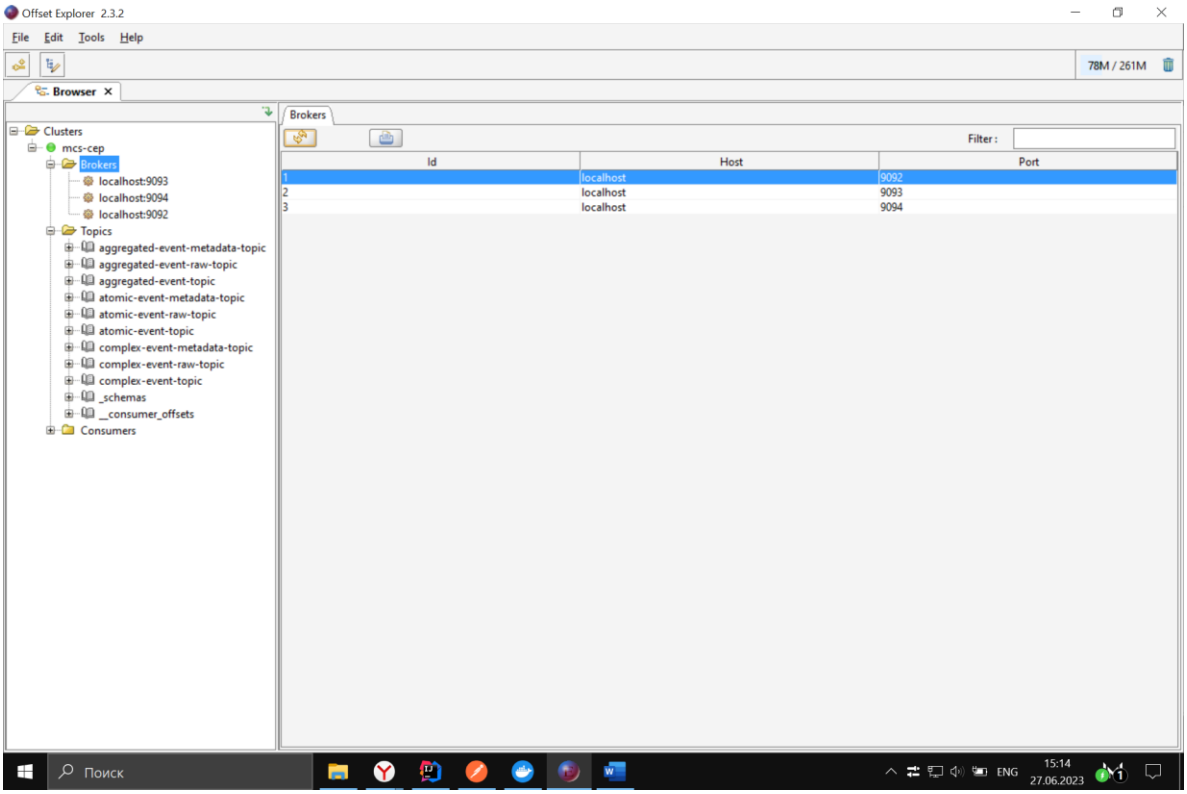

Рисунок В.1– Просмотр состояния кластера с помощью утилиты Offset Explorer (просмотр состояния всех брокеров)

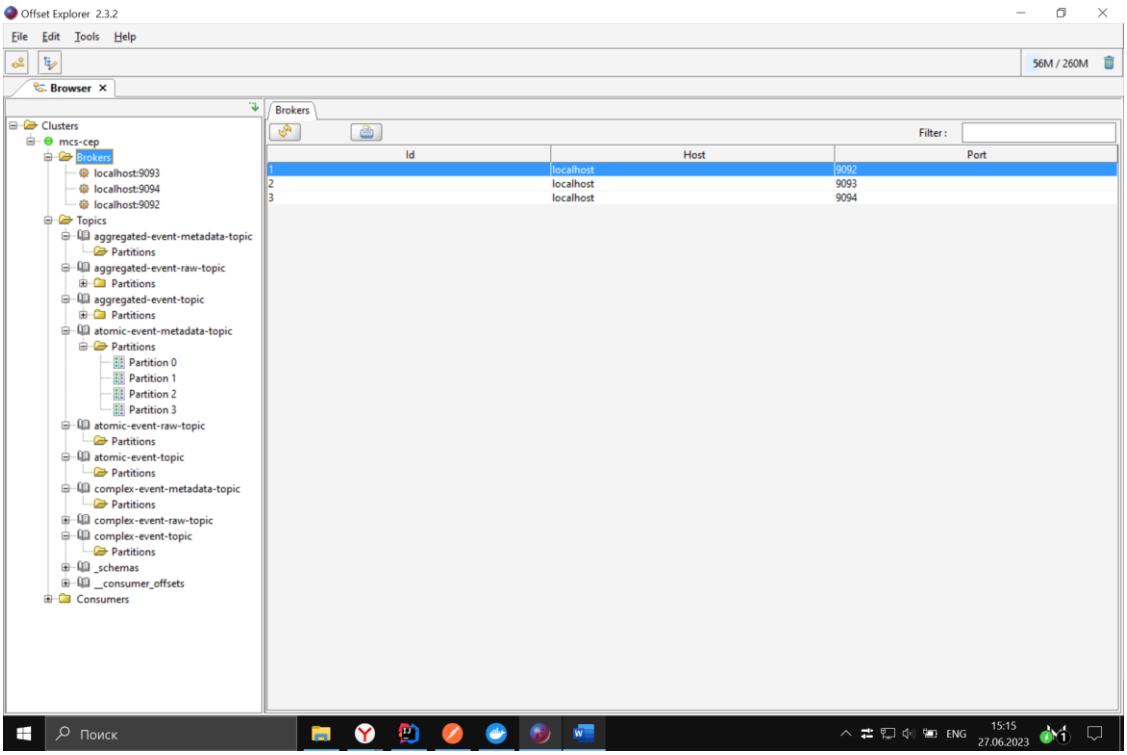

Рисунок В.2 – Просмотр состояния кластера с помощью утилиты Offset Explorer (просмотр состояния тем и их разделов)

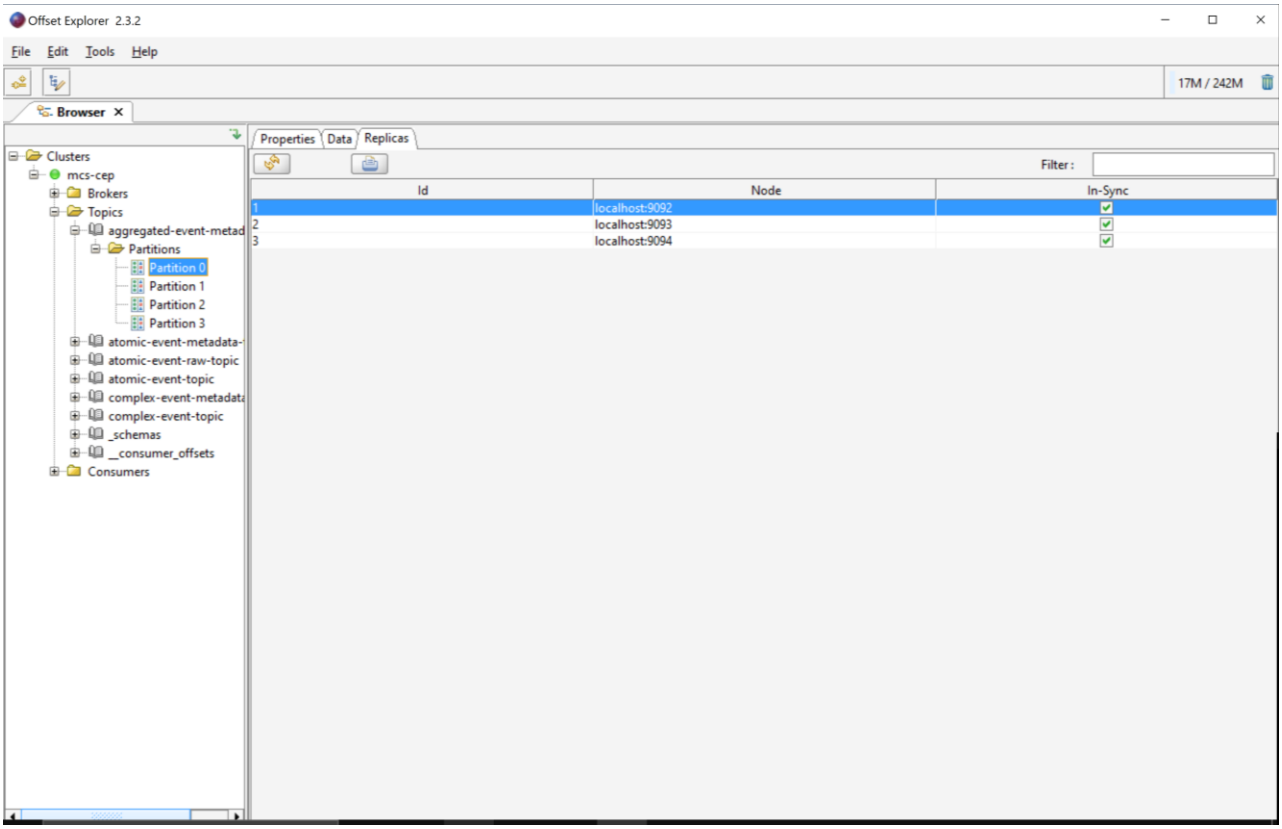

Рисунок В.3 – Просмотр состояния кластера с помощью утилиты Offset Explorer (просмотр состояния репликации)

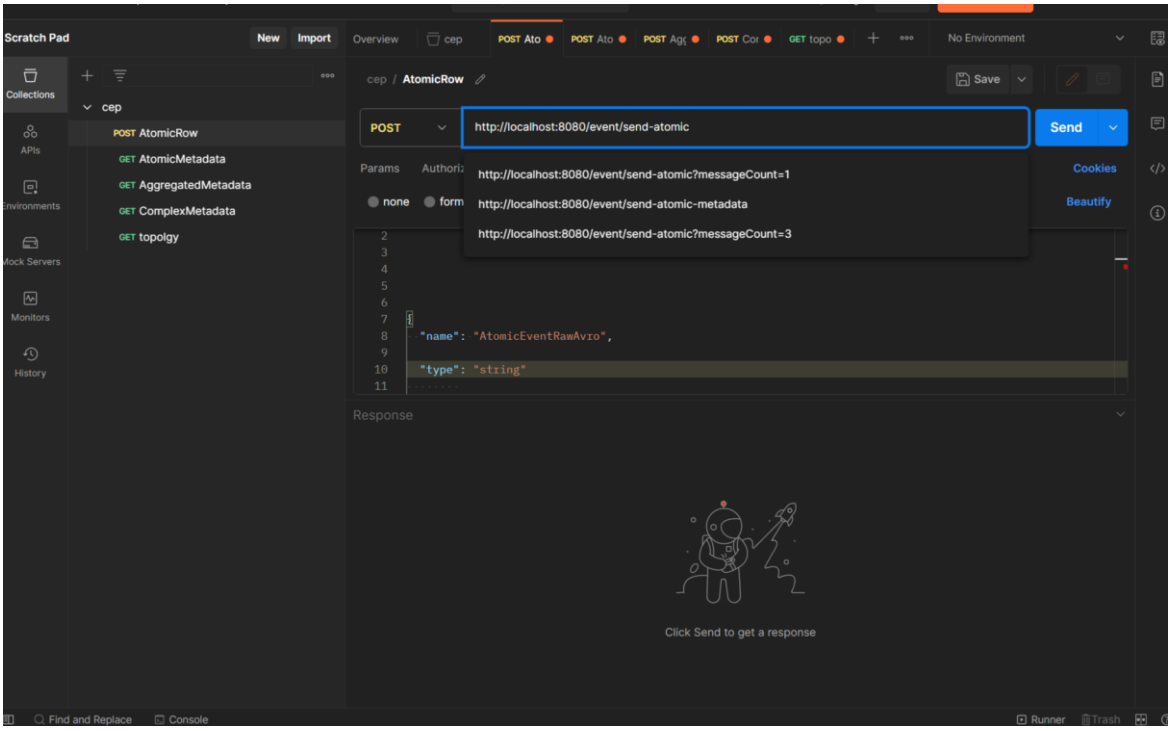

Рисунок В.4 – Отправка сообщений в брокер с помощью утилиты Postman (часть 1)

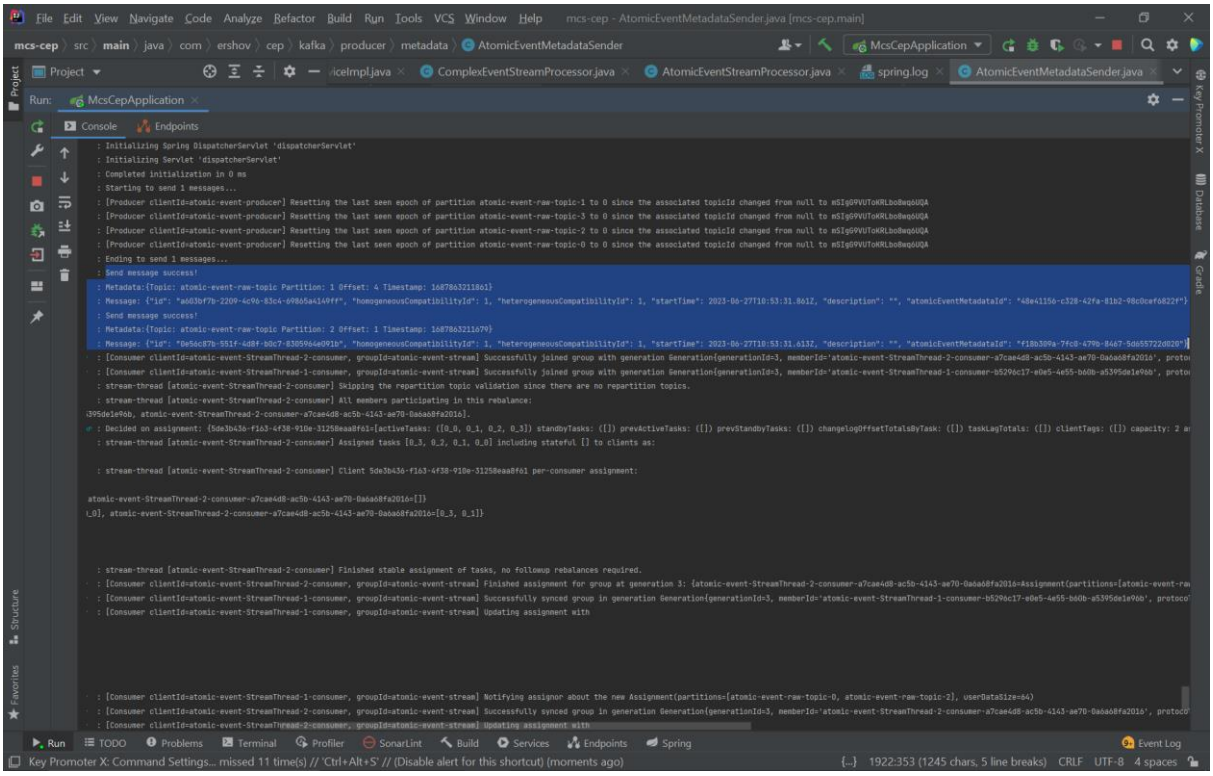

Рисунок В.5 – Отправка сообщений в брокер с помощью утилиты Postman (часть 2)

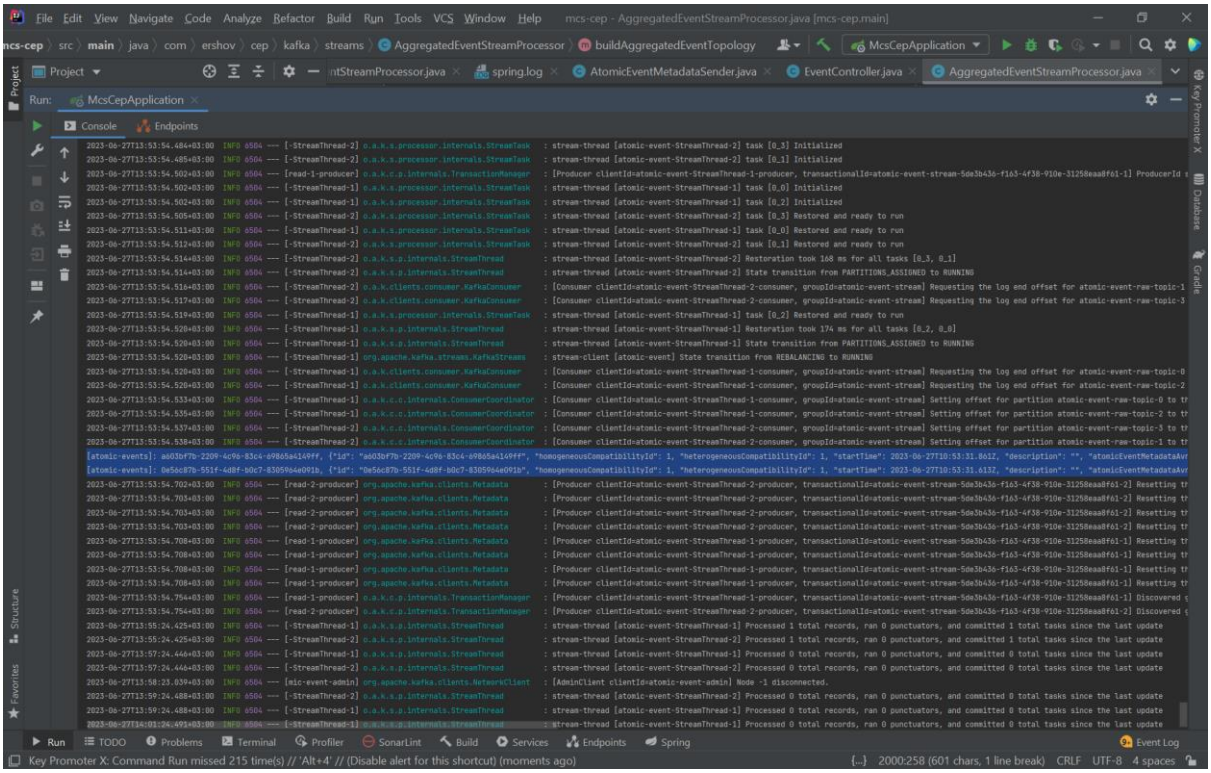

Рисунок В.6 – Обработка события типа Atomic Event по полной топологии для данного процесса с получением результата в консоль (часть 1)

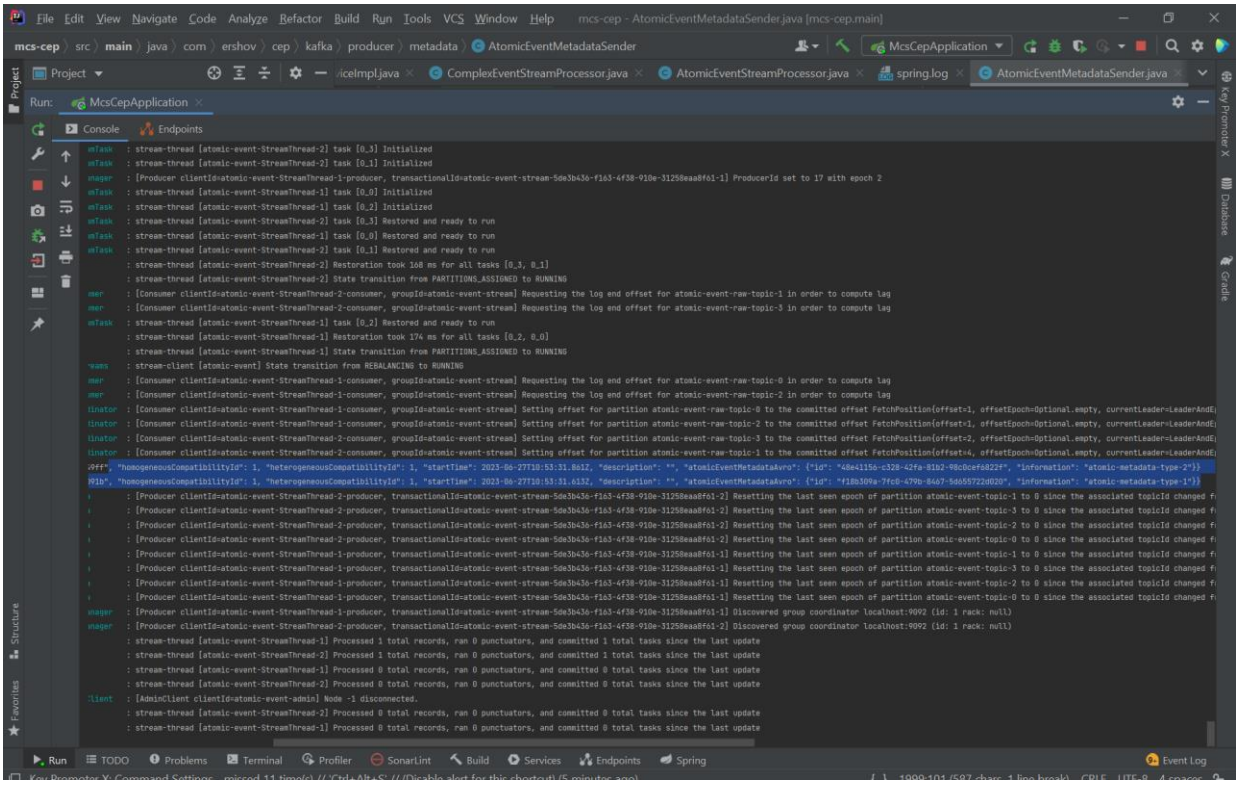

Рисунок В.7 – Обработка события типа Atomic Event по полной топологии для данного процесса с получением результата в консоль (часть 2)

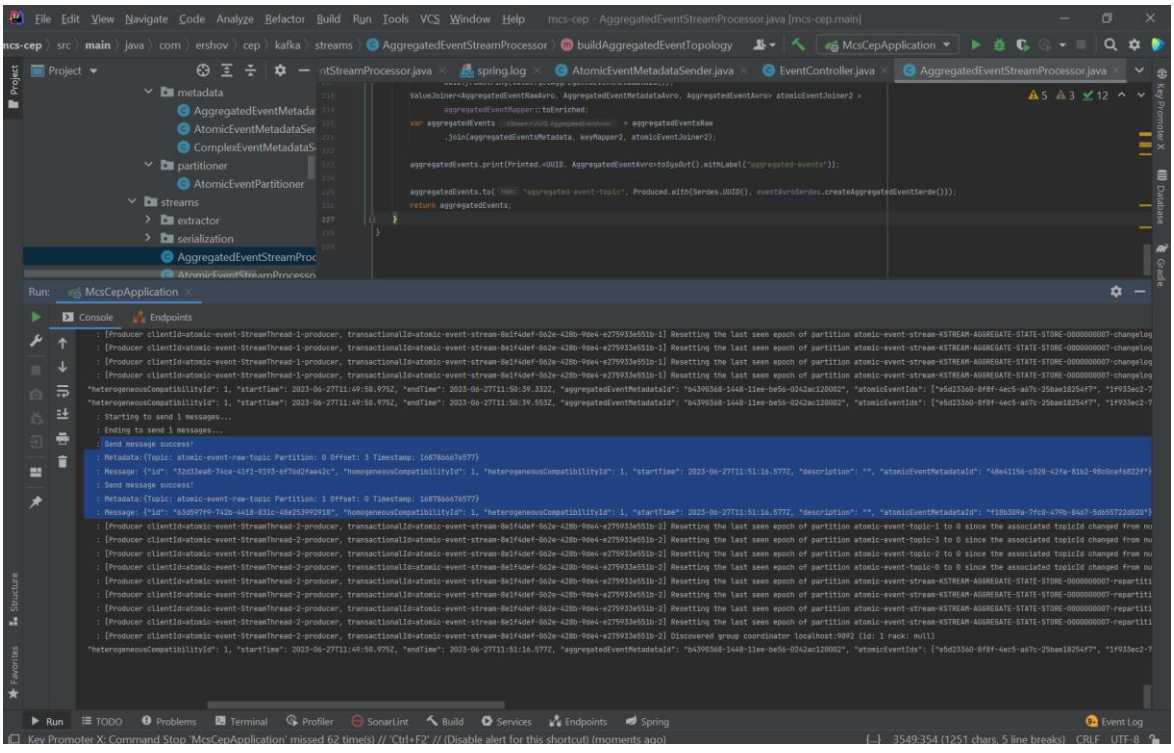

Рисунок В.8 – Обработка события типа Aggregated Event по полной топологии для данного процесса с получением результата в консоль (отправка сообщения)

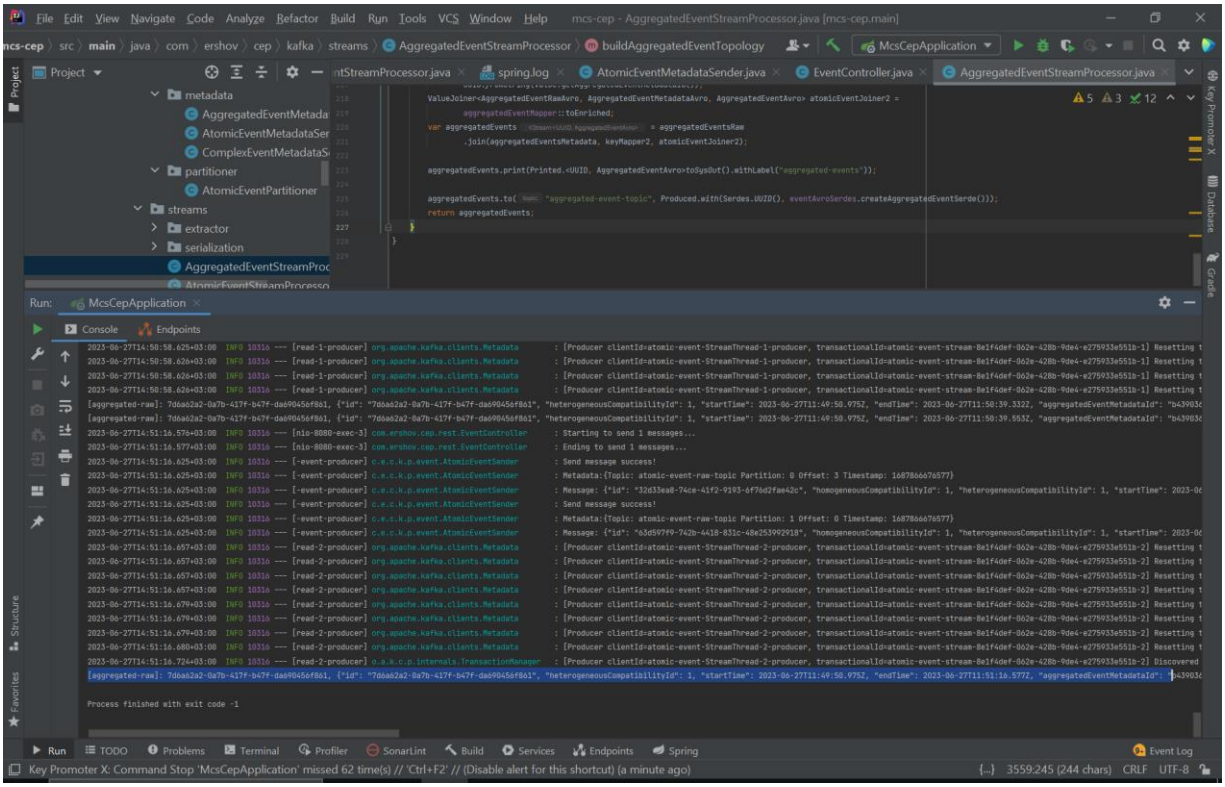

Рисунок В.9 – Обработка события типа Aggregated Event по полной топологии для данного процесса с получением результата в консоль (часть 2)

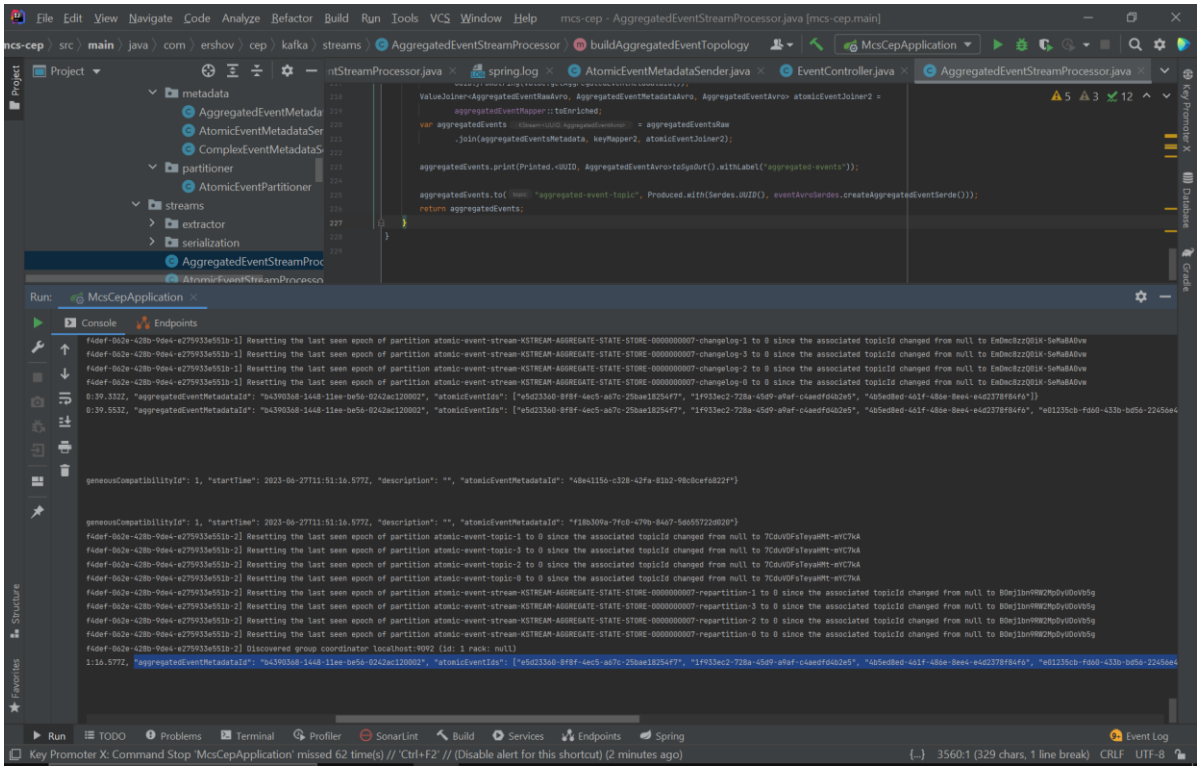

Рисунок В.10 – Обработка события типа Aggregated Event по полной топологии для данного процесса с получением результата в консоль (часть 3)

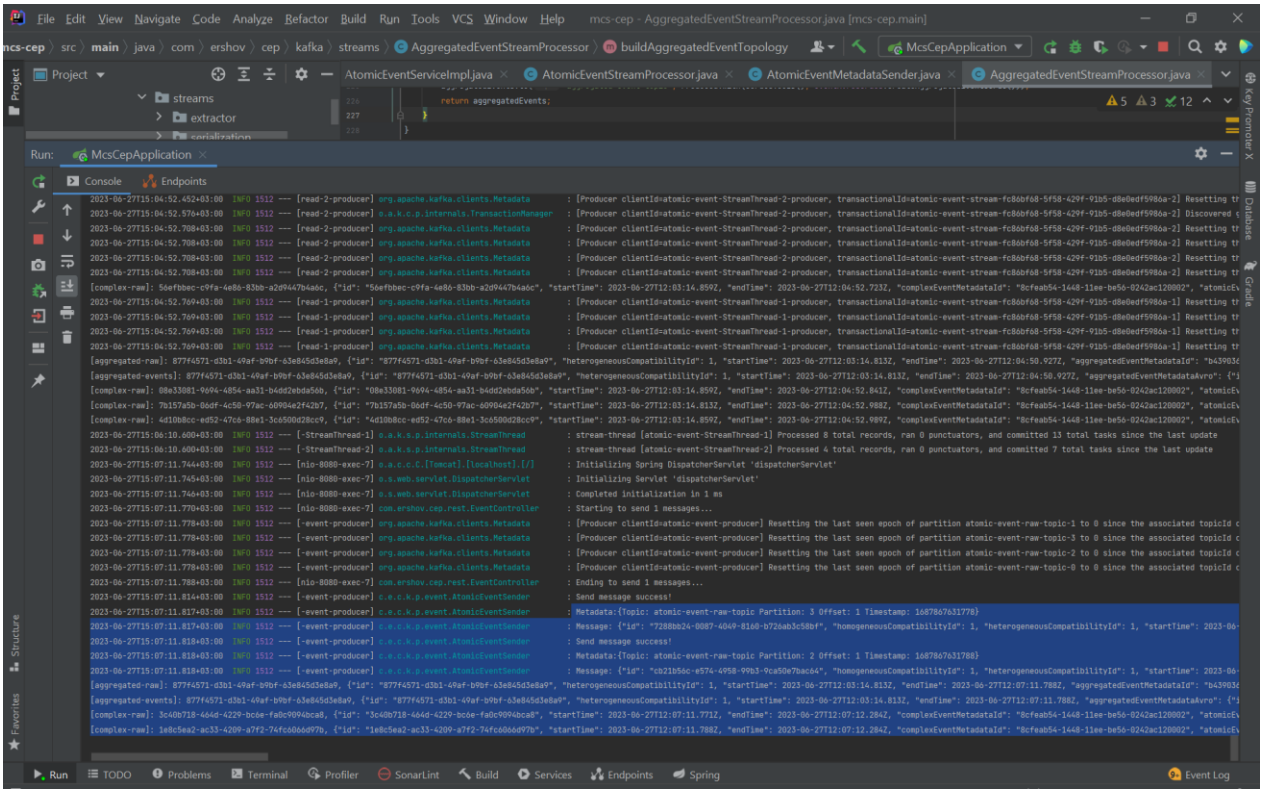

Рисунок В.11 – Обработка события типа Complex Event по полной топологии для данного процесса с получением результата в консоль (часть 1)

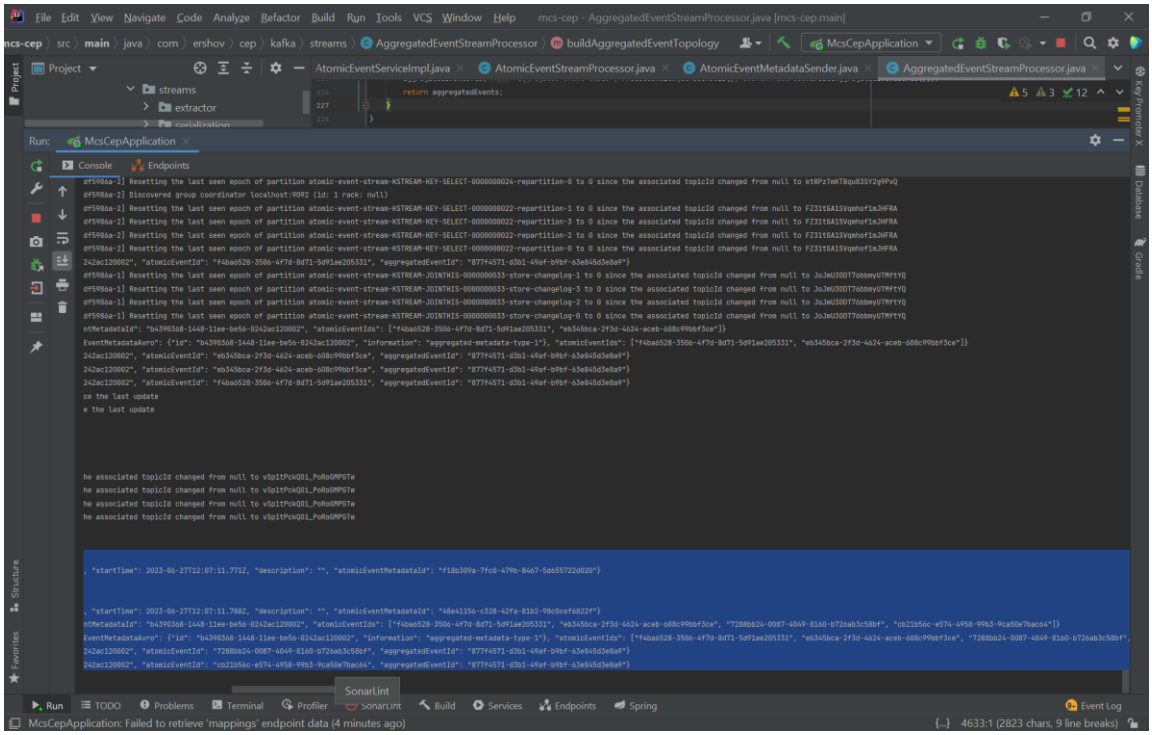

Рисунок В.12 – Обработка события типа Complex Event по полной топологии для данного процесса с получением результата в консоль (часть 2)

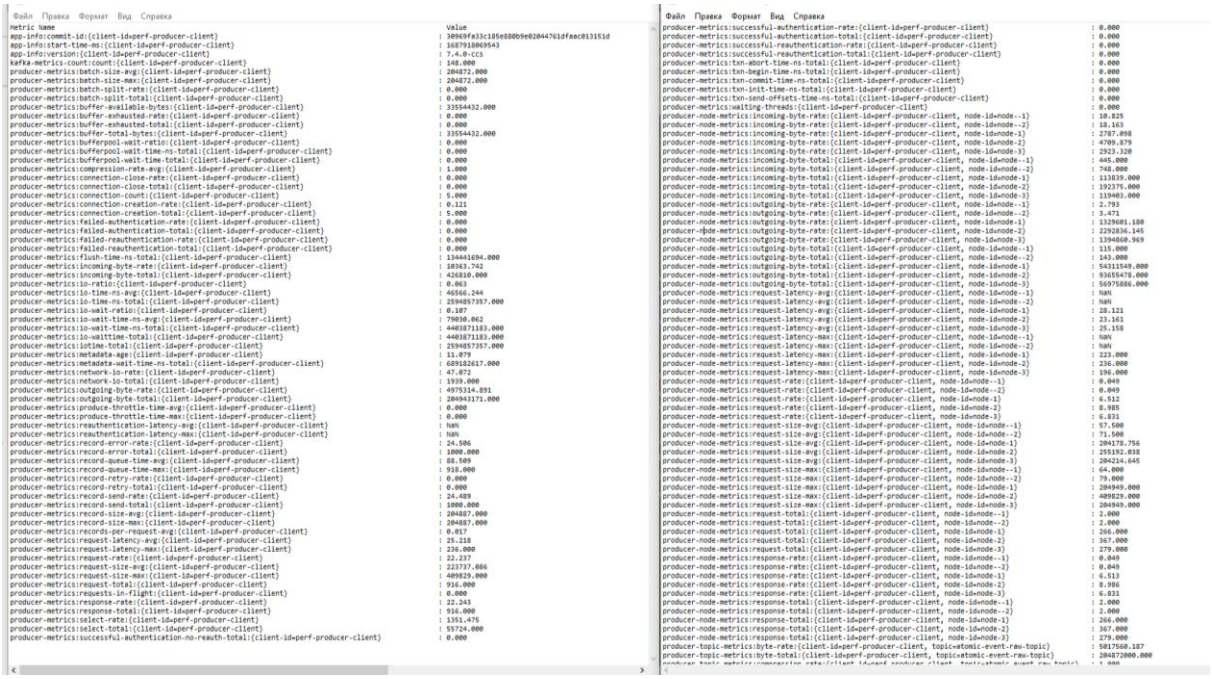

Рисунок В.13 – Метрики нагрузочного тестирование кластера на запись 1000 элементов

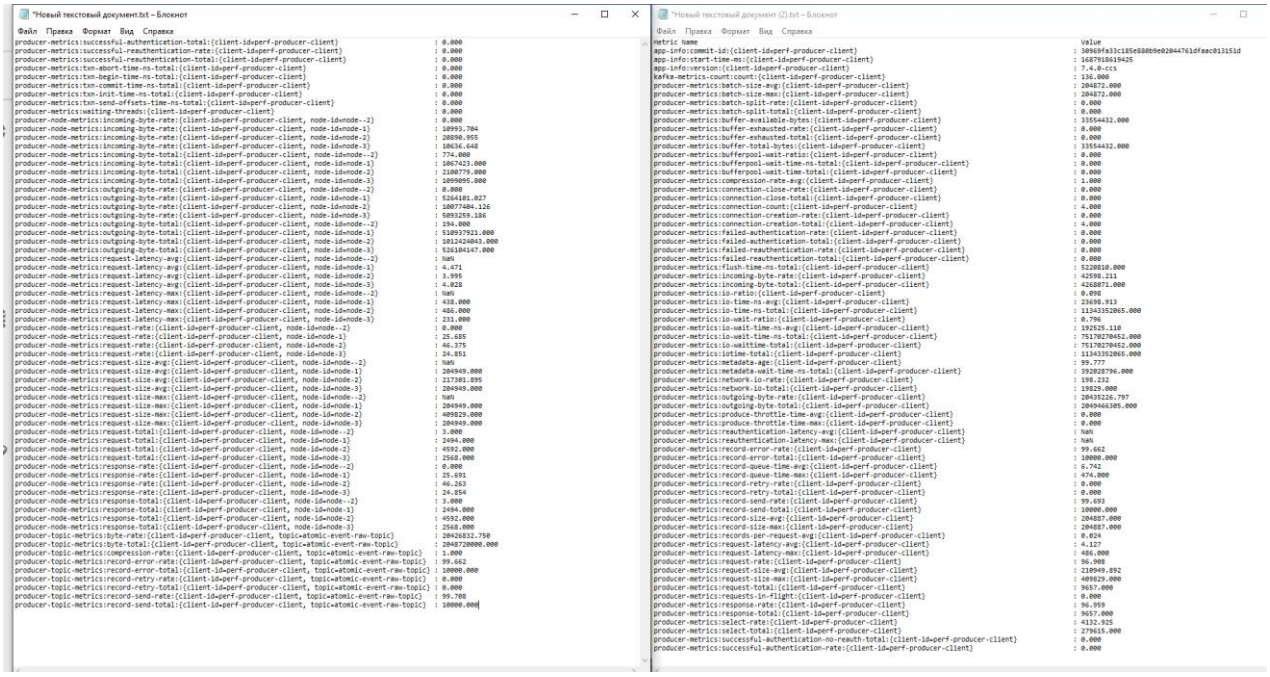

Рисунок В.14 – Метрики нагрузочного тестирование кластера на запись 10000 элементов

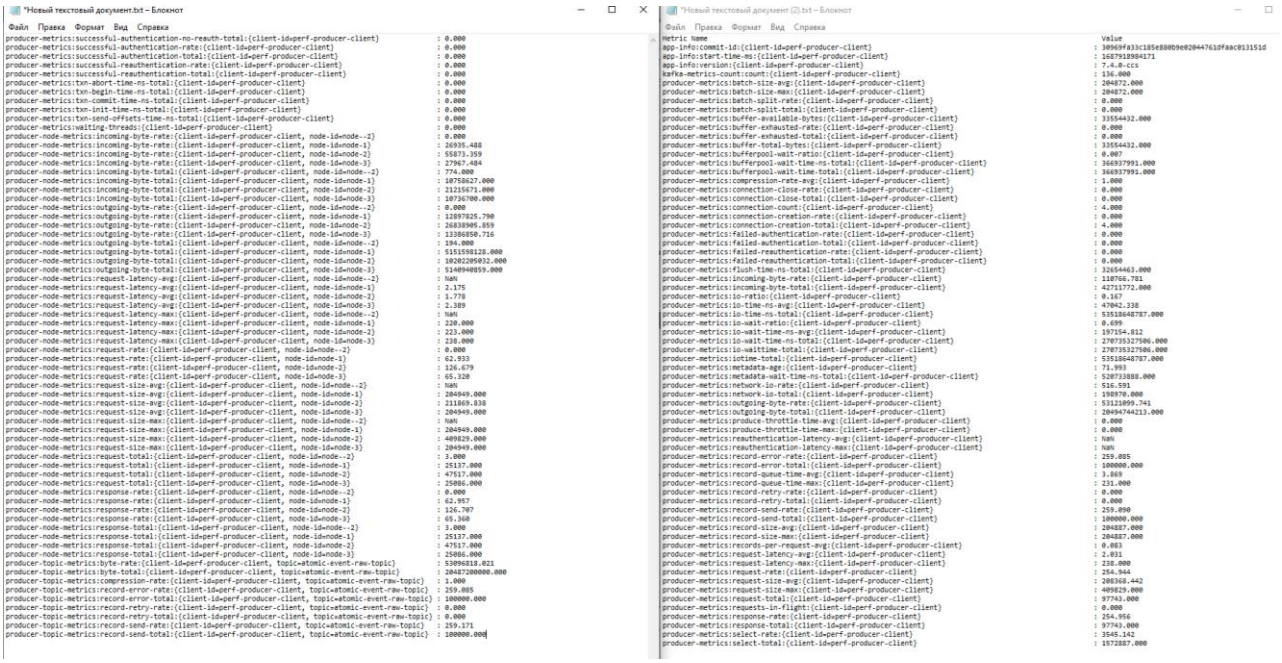

Рисунок В.14 – Метрики нагрузочного тестирование кластера на запись 100000 элементов

Приложение Г

КОДОВАЯ БАЗА

(рекомендуемое)

Листинг Г.1 – Файл KafkaConfig.java

```
@Configuration
public class KafkaConfig {

    @Bean
    @Primary
    public KafkaProperties getKafkaProperties() {
        return new KafkaProperties();
    }
}
```

Листинг Г.2 – Файл KafkaProducerConfig.java

```
@Slf4j
@Configuration
@EnableConfigurationProperties({KafkaProducerSpecificValue.class,
KafkaSchemaValue.class})
public class KafkaProducerConfig {

    private static final String ATOMIC_EVENT_PRODUCER = "atomic-
event-producer";
    private static final String ATOMIC_EVENT_METADATA_PRODUCER =
"atomic-event-metadata-producer";
    private static final String AGGREGATED_EVENT_METADATA_PRODUCER
= "aggregated-event-metadata-producer";
    private static final String COMPLEX_EVENT_METADATA_PRODUCER =
"complex-event-metadata-producer";

    @Bean(destroyMethod = "close")
    public Producer<UUID, AtomicEventRawAvro>
atomicEventProducerConfig(final KafkaProperties kafkaProperties,

final KafkaSchemaValue kafkaSchemaValue,

final KafkaProducerSpecificValue kafkaProducerSpecificValue) {

        var properties = new Properties();
        properties.put(ProducerConfig.CLIENT_ID_CONFIG,
ATOMIC_EVENT_PRODUCER);
        //properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
AtomicEventPartitioner.class);
        properties.putAll(eventProducerConfig(kafkaProperties,
kafkaSchemaValue, kafkaProducerSpecificValue));
        return new KafkaProducer<>(properties);
    }

    @Bean(destroyMethod = "close")
    public Producer<UUID, AtomicEventMetadataAvro>
atomicEventMetadataProducerConfig(final KafkaProperties
kafkaProperties,
```

```
final KafkaSchemaValue kafkaSchemaValue,

final KafkaProducerSpecificValue kafkaProducerSpecificValue) {

        var properties = new Properties();
        properties.put(ProducerConfig.CLIENT_ID_CONFIG,
ATOMIC_EVENT_METADATA_PRODUCER);
        //properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
.class);
        properties.putAll(eventProducerConfig(kafkaProperties,
kafkaSchemaValue, kafkaProducerSpecificValue));
        return new KafkaProducer<>(properties);
    }

    @Bean(destroyMethod = "close")
    public Producer<UUID, AggregatedEventMetadataAvro>
aggregatedEventMetadataProducerConfig(final KafkaProperties
kafkaProperties,

final KafkaSchemaValue kafkaSchemaValue,

final KafkaProducerSpecificValue kafkaProducerSpecificValue) {

        var properties = new Properties();
        properties.put(ProducerConfig.CLIENT_ID_CONFIG,
AGGREGATED_EVENT_METADATA_PRODUCER);
        //properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
.class);
        properties.putAll(eventProducerConfig(kafkaProperties,
kafkaSchemaValue, kafkaProducerSpecificValue));
        return new KafkaProducer<>(properties);
    }

    @Bean(destroyMethod = "close")
    public Producer<UUID, ComplexEventMetadataAvro>
complexEventMetadataProducerConfig(final KafkaProperties
kafkaProperties,

final KafkaSchemaValue kafkaSchemaValue,

final KafkaProducerSpecificValue kafkaProducerSpecificValue) {

        var properties = new Properties();
        properties.put(ProducerConfig.CLIENT_ID_CONFIG,
COMPLEX_EVENT_METADATA_PRODUCER);
        //properties.put(ProducerConfig.PARTITIONER_CLASS_CONFIG,
.class);
        properties.putAll(eventProducerConfig(kafkaProperties,
kafkaSchemaValue, kafkaProducerSpecificValue));
        return new KafkaProducer<>(properties);
    }
```

```
    private Properties eventProducerConfig(final KafkaProperties
kafkaProperties,
                                           final KafkaSchemaValue
kafkaSchemaValue,
                                           final
KafkaProducerSpecificValue kafkaProducerSpecificValue) {
        // --Задачи настройки производителя--
        // Настройки сериализации данных
        // Настройка пакетной отправки
        // Настройка гарантированной доставки
        // Настройка идемпотентность/транзакционность
        // Настройка секционирования
        // Настройка асинхронной отправки
        // deadLetterQueue
        var properties = new Properties();
        properties.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaProperties.getBootstrapServers());
        properties.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
kafkaProperties.getProducer().getKeySerializer());

properties.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
kafkaProperties.getProducer().getValueSerializer());

properties.put(AbstractKafkaSchemaSerDeConfig.SCHEMA_REGISTRY_URL_
CONFIG, kafkaSchemaValue.getSchemaRegistryUrl());

properties.put(AbstractKafkaSchemaSerDeConfig.AUTO_REGISTER_SCHEMA
S, kafkaSchemaValue.getAutoRegisterSchemas());

properties.put(AbstractKafkaSchemaSerDeConfig.USE_LATEST_VERSION,
kafkaSchemaValue.getUseLatestVersion());
        //На данный момент ключ не сериализуется под avro

//properties.put(AbstractKafkaSchemaSerDeConfig.KEY_SUBJECT_NAME_S
TRATEGY, kafkaSchemaRegistryConfig.getKeySubjectNameStrategy());

properties.put(AbstractKafkaSchemaSerDeConfig.VALUE_SUBJECT_NAME_S
TRATEGY, kafkaSchemaValue.getValueSubjectNameStrategy());

//properties.put(KafkaAvroSerializerConfig.AVRO_USE_LOGICAL_TYPE_C
ONVERTERS_CONFIG, true);

        properties.put(ProducerConfig.BATCH_SIZE_CONFIG,
kafkaProducerSpecificValue.getBatchSize());
        properties.put(ProducerConfig.MAX_REQUEST_SIZE_CONFIG,
kafkaProducerSpecificValue.getMaxRequestSize());
        properties.put(ProducerConfig.BUFFER_MEMORY_CONFIG,
kafkaProducerSpecificValue.getBufferMemory());
        properties.put(ProducerConfig.MAX_BLOCK_MS_CONFIG,
kafkaProducerSpecificValue.getMaxBlockMs());
        properties.put(ProducerConfig.LINGER_MS_CONFIG,
```

```
kafkaProducerSpecificValue.getLingerMs());
        properties.put(ProducerConfig.REQUEST_TIMEOUT_MS_CONFIG,
kafkaProducerSpecificValue.getRequestTimeoutMs());
        properties.put(ProducerConfig.RETRY_BACKOFF_MS_CONFIG,
kafkaProducerSpecificValue.getRetryBackoffMs());
        properties.put(ProducerConfig.DELIVERY_TIMEOUT_MS_CONFIG,
kafkaProducerSpecificValue.getDeliveryTimeoutMs());
        properties.put(ProducerConfig.RETRIES_CONFIG,
kafkaProducerSpecificValue.getRetries());
        properties.put(ProducerConfig.ACKS_CONFIG,
kafkaProducerSpecificValue.getAcks());
        properties.put(ProducerConfig.ENABLE_IDEMPOTENCE_CONFIG,
kafkaProducerSpecificValue.getEnableIdempotence());

properties.put(ProducerConfig.MAX_IN_FLIGHT_REQUESTS_PER_CONNECTIO
N,
kafkaProducerSpecificValue.getMaxInFlightRequestsPerConnection());
        return properties;
    }
}
```

Листинг Г.3 – Файл KafkaStreamsConfig.java

```
@Slf4j
@EnableKafka
@EnableKafkaStreams
@Configuration
@EnableConfigurationProperties({
        KafkaStreamsConsumerValue.class,
        KafkaStreamsSpecificValue.class
})
public class KafkaStreamsConfig {

    @Bean(name =
KafkaStreamsDefaultConfiguration.DEFAULT_STREAMS_CONFIG_BEAN_NAME)
    public KafkaStreamsConfiguration kafkaStreamsConfig(final
KafkaProperties kafkaProperties,
                                                        final
KafkaStreamsConsumerValue kafkaStreamsConsumerValue,
                                                        final
KafkaStreamsSpecificValue kafkaSpecificStreams) {

        var properties = new HashMap<String, Object>();
        properties.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG,
kafkaProperties.getBootstrapServers());
        properties.put(StreamsConfig.CLIENT_ID_CONFIG,
kafkaProperties.getStreams().getClientId());
        properties.put(StreamsConfig.APPLICATION_ID_CONFIG,
kafkaProperties.getStreams().getApplicationId());


properties.putAll(streamsConsumerConfig(kafkaStreamsConsumerValue)
```

```
);

properties.putAll(streamsSpecificConfig(kafkaSpecificStreams));
        //TODO: заполнить конфиги

//properties.putAll(streamsMainConsumerConfig(kafkaStreamsConsumer
Value));

//properties.putAll(streamsGlobalConsumerConfig(kafkaStreamsConsum
erValue));
        return new KafkaStreamsConfiguration(properties);
    }

    /**
     * Конфигурация всех потребителей, использующихся kstreams
     *
     * @param kafkaStreamsConsumerValue настройки стандартного
потребителя
     * @return конфигурация
     */
    private Map<String, Object>
streamsConsumerConfig(KafkaStreamsConsumerValue
kafkaStreamsConsumerValue) {
        // --Задачи настройки потребителя--
        // Настройки десериализации данных
        // Настройка лимитов читаемых данных
        // Настройка тайм-аутов потребителя во избежании
перебалансировки!
        // Настройка стратегии назначения разделов(влияет на
перебалансировку)
        // Настройка фиксации смещения при чтении
        // Настройка многопоточной обработки(делается внутри
логики)
        // Настройка консистентного приема(делается внутри логики)
        var properties = new HashMap<String, Object>();
        // Не требуется(!)

//properties.put(StreamsConfig.mainConsumerPrefix(ConsumerConfig.C
LIENT_ID_CONFIG), kafkaStreamsConsumerConfig.getClientId());

//properties.put(StreamsConfig.mainConsumerPrefix(ConsumerConfig.G
ROUP_ID_CONFIG), kafkaStreamsConsumerConfig.getGroupId());

//properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.CLIEN
T_RACK_CONFIG),
kafkaStreamConsumerSpecificConfig.getClientRack());

//properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.GROUP
_INSTANCE_ID_CONFIG),
kafkaStreamsConsumerValue.getGroupInstanceId());
        properties.put("key.converter.schema.registry.url",
"http://localhost:8081");
```

```
        properties.put("value.converter.schema.registry.url",
"http://localhost:8081");
        properties.put("schema.registry.url",
"http://localhost:8081");


properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.FETCH_M
IN_BYTES_CONFIG), kafkaStreamsConsumerValue.getFetchMinBytes());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.FETCH_M
AX_BYTES_CONFIG), kafkaStreamsConsumerValue.getFetchMaxBytes());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.MAX_POL
L_RECORDS_CONFIG), kafkaStreamsConsumerValue.getMaxPollRecords());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.MAX_PAR
TITION_FETCH_BYTES_CONFIG),
kafkaStreamsConsumerValue.getMaxPartitionFetchBytes());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.FETCH_M
AX_WAIT_MS_CONFIG),
kafkaStreamsConsumerValue.getFetchMaxWaitMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.HEARTBE
AT_INTERVAL_MS_CONFIG),
kafkaStreamsConsumerValue.getHeartbeatIntervalMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.SESSION
_TIMEOUT_MS_CONFIG),
kafkaStreamsConsumerValue.getSessionTimeoutMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.MAX_POL
L_INTERVAL_MS_CONFIG),
kafkaStreamsConsumerValue.getMaxPollIntervalMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.AUTO_CO
MMIT_INTERVAL_MS_CONFIG),
kafkaStreamsConsumerValue.getAutoCommitIntervalMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.REQUEST
_TIMEOUT_MS_CONFIG),
kafkaStreamsConsumerValue.getRequestTimeoutMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.RETRY_B
ACKOFF_MS_CONFIG), kafkaStreamsConsumerValue.getRetryBackoffMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.DEFAULT
_API_TIMEOUT_MS_CONFIG),
kafkaStreamsConsumerValue.getDefaultApiTimeoutMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.RECONNE
CT_BACKOFF_MS_CONFIG),
```

```
kafkaStreamsConsumerValue.getReconnectBackoffMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.RECONNE
CT_BACKOFF_MAX_MS_CONFIG),
kafkaStreamsConsumerValue.getReconnectBackoffMaxMs());

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.AUTO_OF
FSET_RESET_CONFIG),
kafkaStreamsConsumerValue.getAutoOffsetReset());
        // Недоступно для переопределения(!)

//properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.ENABL
E_AUTO_COMMIT_CONFIG),
kafkaStreamsConsumerValue.getEnableAutoCommit());
        // НЕ ставить StreamPartitioner!!!

properties.put(StreamsConfig.consumerPrefix(ConsumerConfig.PARTITI
ON_ASSIGNMENT_STRATEGY_CONFIG),
kafkaStreamsConsumerValue.getPartitionAssignmentStrategy());
        return properties;
    }

    /**
     * Конфигурация main-потребителя, использующихся kstreams
     *
     * @param kafkaStreamsConsumerValue настройки стандартного
потребителя
     * @return конфигурация
     */
    private Map<String, Object>
streamsMainConsumerConfig(KafkaStreamsConsumerValue
kafkaStreamsConsumerValue) {

        var properties = new HashMap<String, Object>();
        // НЕ указывать StreamsPartitionAssignor, параметр
устанавливается автоматически и запрещает даже правильное
переопределение!
        return properties;
    }

    /**
     * Конфигурация global-потребителя, использующегося kstreams
     *
     * @param kafkaStreamsConsumerValue настройки стандартного
потребителя
     * @return конфигурация
     */
    private Map<String, Object>
streamsGlobalConsumerConfig(KafkaStreamsConsumerValue
kafkaStreamsConsumerValue) {

        var properties = new HashMap<String, Object>();
```

```
        return properties;
    }

    /**
     * Конфигурация kstreams
     *
     * @param kafkaStreamsSpecificValue настройки kstreams
     * @return
     */
    private Map<String, Object>
streamsSpecificConfig(KafkaStreamsSpecificValue
kafkaStreamsSpecificValue) {
        // --Задачи настройки kstreams--
        // Настройки сериализации/десериализации данных
        // Настройка гарантированной обработки
        // Настройка многопоточной обработки(разделы->задач-
>потоки)
        // Оптимизация данных на пути по топологии(буфферизация)
        // Настройка доступности/отказоустойчивости
состояний(репликация,тайм-ауты по перебалансировке)
        // Настройка RocksDB
        var properties = new HashMap<String, Object>();

properties.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG,
kafkaStreamsSpecificValue.getDefaultKeySerde());

properties.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG,
kafkaStreamsSpecificValue.getDefaultValueSerde());

        properties.put(StreamsConfig.MAX_TASK_IDLE_MS_CONFIG,
kafkaStreamsSpecificValue.getMaxTaskIdleMs());
        properties.put(StreamsConfig.TASK_TIMEOUT_MS_CONFIG,
kafkaStreamsSpecificValue.getTaskTimeoutMs());
        properties.put(StreamsConfig.COMMIT_INTERVAL_MS_CONFIG,
kafkaStreamsSpecificValue.getCommitIntervalMs());
        properties.put(StreamsConfig.METADATA_MAX_AGE_CONFIG,
kafkaStreamsSpecificValue.getMetadataMaxAgeMs());

properties.put(StreamsConfig.BUFFERED_RECORDS_PER_PARTITION_CONFIG
, kafkaStreamsSpecificValue.getBufferedRecordsPerPartition());

properties.put(StreamsConfig.CACHE_MAX_BYTES_BUFFERING_CONFIG,
kafkaStreamsSpecificValue.getCacheMaxBytesBuffering());

properties.put(StreamsConfig.REPARTITION_PURGE_INTERVAL_MS_CONFIG,
kafkaStreamsSpecificValue.getRepartitionPurgeIntervalMs());
        properties.put(StreamsConfig.TOPOLOGY_OPTIMIZATION_CONFIG,
kafkaStreamsSpecificValue.getTopologyOptimization());
        //processing.guarantee переопределяет несколько параметров
внутренних производителей и потребителей
        properties.put(StreamsConfig.PROCESSING_GUARANTEE_CONFIG,
kafkaStreamsSpecificValue.getProcessingGuarantee());
```

```
properties.put(StreamsConfig.DEFAULT_DESERIALIZATION_EXCEPTION_HAN
DLER_CLASS_CONFIG,
kafkaStreamsSpecificValue.getDefaultDeserializationExceptionHandle
r());

properties.put(StreamsConfig.DEFAULT_PRODUCTION_EXCEPTION_HANDLER_
CLASS_CONFIG,
kafkaStreamsSpecificValue.getDefaultProductionExceptionHandler());
        properties.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG,
kafkaStreamsSpecificValue.getNumStreamThreads());
        properties.put(StreamsConfig.DEFAULT_DSL_STORE_CONFIG,
kafkaStreamsSpecificValue.getDefaultDslStore());
        //properties.put(StreamsConfig.STATE_DIR_CONFIG,
kafkaStreamsSpecificValue.getStateDir());

properties.put(StreamsConfig.STATESTORE_CACHE_MAX_BYTES_CONFIG,
kafkaStreamsSpecificValue.getStatestoreCacheMaxBytes());

properties.put(StreamsConfig.ACCEPTABLE_RECOVERY_LAG_CONFIG,
kafkaStreamsSpecificValue.getAcceptableRecoveryLag());
        properties.put(StreamsConfig.REPLICATION_FACTOR_CONFIG,
kafkaStreamsSpecificValue.getReplicationFactor());
        properties.put(StreamsConfig.NUM_STANDBY_REPLICAS_CONFIG,
kafkaStreamsSpecificValue.getNumStandbyReplicas());
        properties.put(StreamsConfig.MAX_WARMUP_REPLICAS_CONFIG,
kafkaStreamsSpecificValue.getMaxWarmupReplicas());

properties.put(StreamsConfig.PROBING_REBALANCE_INTERVAL_MS_CONFIG,
kafkaStreamsSpecificValue.getProbingRebalanceIntervalMs());

properties.put(StreamsConfig.STATE_CLEANUP_DELAY_MS_CONFIG,
kafkaStreamsSpecificValue.getStateCleanupDelayMs());
        properties.put(StreamsConfig.APPLICATION_SERVER_CONFIG,
kafkaStreamsSpecificValue.getApplicationServer());

//properties.put(StreamsConfig.ROCKSDB_CONFIG_SETTER_CLASS_CONFIG,
kafkaStreamsSpecificValue.getRocksdbConfigSetter());

properties.put(StreamsConfig.METRICS_RECORDING_LEVEL_CONFIG,
kafkaStreamsSpecificValue.getMetricsRecordingLevel());
        //properties.put(StreamsConfig.UPGRADE_FROM_CONFIG,
kafkaStreamsSpecificValue.getUpgradeFrom());
        return properties;
    }
}
```

Листинг Г.4 – Файл AggregatedEventStreamProcessor.java

```
@Slf4j
@Component
@RequiredArgsConstructor
```

```
public class AggregatedEventStreamProcessor {

    @Value("${spring.kafka.topics.atomic}")
    private String atomicEventTopic;

    @Value("${spring.kafka.topics.aggregated-raw}")
    private String aggregatedEventRawTopic;

    @Value("${spring.kafka.topics.aggregated")
    private String aggregatedEventTopic;

    @Value("${spring.kafka.topics.aggregated-metadata}")
    private String aggregatedEventMetadataTopic;

    private final EventAvroSerdes eventAvroSerdes;

    private final AggregatedEventMapper aggregatedEventMapper;

    private final AtomicEventMapper atomicEventMapper;

    private final AggregatedEventService aggregatedEventService;


@Bean
    public KStream<UUID, AggregatedEventRawAvro>
buildAggregatedEventsRawTopology(StreamsBuilder streamsBuilder) {

        //1. На вход подаются атомарные события
        var atomicEvents = streamsBuilder.stream(
                atomicEventTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventSerde())
        );

        //2. Атомарные события превращаются в агрегированные
        // Группировка по ключу совместности, в данном случае
происходит секционирование, которое сопровождается
        // созданием нового топика, где события с одинаковыми
ключами распределяются по одинаковым разделам
        var groupedAtomicEvent = atomicEvents.groupBy(
                (key, value) ->
value.getHomogeneousCompatibilityId(),
                Grouped.with(Serdes.Integer(),
eventAvroSerdes.createAtomicEventSerde()));
        // Инициализатор - объект, который будет создавать при
каждом появлении нового ключа из groupedAtomicEvent
        Initializer<AggregatedEventRawAvro>
aggregatedEventRawInitializer = () ->
AggregatedEventRawAvro.newBuilder()
                .setId(UUID.randomUUID().toString())
                .setAtomicEventIds(new LinkedList<>())
                .build();
```

```
        // Агрегатор - метод, определяющий логику агрегирования
объектов
        Aggregator<Integer, AtomicEventAvro,
AggregatedEventRawAvro> aggregatedEventRawAggregator =
                (key, value, aggregate) -> {
                    var startTimeAtomic = value.getStartTime();
                    var startTimeAggregated =
aggregate.getStartTime();
                    var endTimeAggregated =
aggregate.getEndTime();
                    if (startTimeAggregated == null
                            ||
startTimeAggregated.isAfter(startTimeAtomic)) {
                        aggregate.setStartTime(startTimeAtomic);
                    }
                    if (endTimeAggregated == null
                            ||
endTimeAggregated.isBefore(startTimeAtomic)) {

aggregate.setEndTime(value.getStartTime());
                    }
                    var atomicEventIds =
aggregate.getAtomicEventIds();
                    atomicEventIds.add(key.toString());

aggregate.setHeterogeneousCompatibilityId(value.getHeterogeneousCo
mpatibilityId());

aggregate.setAggregatedEventMetadataId("3e66e456-80db-40dc-8cfd-
13796fa00e6b");
                    return aggregate;
                };
        // Агрегация
        var aggregatedEventsRaw =
                groupedAtomicEvent.aggregate(
                        aggregatedEventRawInitializer,
                        aggregatedEventRawAggregator
                )
                        .toStream()
                        .map((key, value) ->
KeyValue.pair(UUID.fromString(value.getId()), value));

        //3. Далее запись в тему
        aggregatedEventsRaw.to(aggregatedEventRawTopic,
Produced.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventRawSerde()));

        return aggregatedEventsRaw;
    }

    @Bean
    public KStream<UUID, AggregatedEventAvro>
```

```
buildAggregatedEventTopology(StreamsBuilder streamsBuilder) {

        //1. На вход подаются сырые агрегированные события,
которые должны обогащаться метаданными
        var aggregatedEventsRaw = streamsBuilder.stream(
                aggregatedEventRawTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventRawSerde())
        );
        //2. Создается глобальное состояние(копируется для всех
разделов) метаданных агрегированных события
        var aggregatedEventsMetadata = streamsBuilder.globalTable(
                aggregatedEventMetadataTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventMetadataSerde())
        );

        // 1. Обогащение агрегированных событий
        // Сопоставление ключей на основании получаемого объекта
из соединения
        // [тип ключа потока основного потока|тип значения
основного потока|тип ключа просматриваемого потока ]
        KeyValueMapper<UUID, AggregatedEventRawAvro, UUID>
keyMapper = (key, value) ->

UUID.fromString(value.getAggregatedEventMetadataId());
        // Соединитель между потоками сырого агрегированного
события и метаданными агрегированного события
        ValueJoiner<AggregatedEventRawAvro,
AggregatedEventMetadataAvro, AggregatedEventAvro>
atomicEventJoiner =
                aggregatedEventMapper::toEnriched;
        var aggregatedEvents = aggregatedEventsRaw
                .join(aggregatedEventsMetadata, keyMapper,
atomicEventJoiner);

        //4. Отвлеченная бизнес логика(вывод на консоль)
        //TODO: добавить сервис
        aggregatedEvents.print(Printed.<UUID,
AggregatedEventAvro>toSysOut().withLabel("aggregated-events"));

        //5. Далее записываем в топик
        aggregatedEvents.to(aggregatedEventTopic,
Produced.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventSerde()));

        return aggregatedEvents;
    }
```

Листинг Г.5 – Файл AtomicEventStreamProcessor.java

```
@Slf4j
@Component
@RequiredArgsConstructor
public class AtomicEventStreamProcessor {

    @Value("${spring.kafka.topics.atomic-raw}")
    private String atomicEventRawTopic;

    @Value("${spring.kafka.topics.atomic}")
    private String atomicEventTopic;

    @Value("${spring.kafka.topics.atomic-metadata}")
    private String atomicEventMetadataTopic;

    private final EventAvroSerdes eventAvroSerdes;

    private final AtomicEventMapper atomicEventMapper;

    private final AtomicEventService atomicEventService;


   @Bean
    public KStream<UUID, AtomicEventAvro>
buildAtomicEventsTopology(StreamsBuilder streamsBuilder) {

        //1. На вход подаются сырые атомарные события, которые
должны обогащаться метаданными
        var atomicEventsRaw = streamsBuilder.stream(
                atomicEventRawTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventRawSerde())
        );
        //2. Создается глобальное состояние(копируется для всех
разделов) метаданных атомарного события
        var atomicEventsMetadata = streamsBuilder.globalTable(
                atomicEventMetadataTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventMetadataSerde())
        );

        //3. Обогащение атомарных событий
        // Сопоставление ключей на основании получаемого объекта
из соединения
        // [тип ключа потока основного потока|тип значения
основного потока|тип ключа просматриваемого потока ]
        KeyValueMapper<UUID, AtomicEventRawAvro, UUID> keyMapper =
(key, value) ->
                UUID.fromString(value.getAtomicEventMetadataId());
        // Соединитель между потоками сырого атомарного события и
метаданными атомарного события
        ValueJoiner<AtomicEventRawAvro, AtomicEventMetadataAvro,
AtomicEventAvro> atomicEventJoiner =
```

```
                atomicEventMapper::toEnriched;
        var atomicEvents = atomicEventsRaw
                .join(atomicEventsMetadata, keyMapper,
atomicEventJoiner);

        //4. Отвлеченная бизнес логика(вывод на консоль)
        //TODO: добавить сервис
        atomicEvents.print(Printed.<UUID,
AtomicEventAvro>toSysOut().withLabel("atomic-events"));

        //5. Далее записываем в топик
        atomicEvents.to(atomicEventTopic,
Produced.with(Serdes.UUID(),eventAvroSerdes.createAtomicEventSerde
()));

        return atomicEvents;
    }
}
```

Листинг Г.6 – Файл ComplexEventStreamProcessor.java

```
@Slf4j
@Component
@RequiredArgsConstructor
public class ComplexEventStreamProcessor {

    @Value("${spring.kafka.topics.atomic}")
    private String atomicEventTopic;

    @Value("${spring.kafka.topics.aggregated}")
    private String aggregatedEventTopic;

    @Value("${spring.kafka.topics.complex-raw")
    private String complexEventRawTopic;

    @Value("${spring.kafka.topics.complex")
    private String complexEventTopic;

    @Value("${spring.kafka.topics.complex-metadata}")
    private String complexEventMetadataTopic;

    private final EventAvroSerdes eventAvroSerdes;

    private final ComplexEventMapper complexEventMapper;

    private final ComplexEventService complexEventService;

    private final AtomicEventMapper atomicEventMapper;


    @Value("${spring.kafka.topics.aggregated-metadata}")
    private String aggregatedEventMetadataTopic;
```

```
    private final AggregatedEventMapper aggregatedEventMapper;



    @Bean
    public KStream<UUID, ComplexEventRawAvro>
buildComplexEventsRawTopology(StreamsBuilder streamsBuilder) {

        //1. На вход подаются атомарные и агрегированные события
        var atomicEvents = streamsBuilder.stream(
                atomicEventTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAtomicEventSerde())
                        .withTimestampExtractor(new
AtomicEventTimestampExtractor())
        );
        var aggregatedEvents = streamsBuilder.stream(
                aggregatedEventTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createAggregatedEventSerde())
                        .withTimestampExtractor(new
AggregatedEventTimestampExtractor())
        );

        // Создание скользящего окна с периодом отсрочки попадания
в него
        var joinWindows = JoinWindows.ofTimeDifferenceAndGrace(
                Duration.ofSeconds(5),
                Duration.ofSeconds(2)
        );

        // Создание параметров объединения
        var joinParams = StreamJoined
                .with(
                        Serdes.UUID(),

eventAvroSerdes.createAggregatedEventSerde(),
                        eventAvroSerdes.createAtomicEventSerde()
                );

        ValueJoiner<AggregatedEventAvro, AtomicEventAvro,
ComplexEventRawAvro> valueJoiner =
                (aggregatedEventAvro, atomicEventAvro) -> {
                    var complexEventRawAvro =
ComplexEventRawAvro.newBuilder()
                            .setId(UUID.randomUUID().toString())

.setAtomicEventId(atomicEventAvro.getId())

.setAggregatedEventId(aggregatedEventAvro.getId())
```

```
                                .build();
                        var atomicEventStartTime =
atomicEventAvro.getStartTime();
                        var aggregatedEventStartTime =
aggregatedEventAvro.getStartTime();
                        complexEventRawAvro.setStartTime(

aggregatedEventStartTime.isBefore(atomicEventStartTime)
                                        ? aggregatedEventStartTime :
atomicEventStartTime
                        );
                        var aggregatedEventEndTime =
aggregatedEventAvro.getStartTime();
                        complexEventRawAvro.setStartTime(

aggregatedEventEndTime.isAfter(atomicEventStartTime)
                                        ? aggregatedEventEndTime :
atomicEventStartTime
                        );
                        return complexEventRawAvro;
                };

        var complexEventsRaw = aggregatedEvents
                .join(atomicEvents, valueJoiner, joinWindows,
joinParams);

        complexEventsRaw.to(complexEventRawTopic,
Produced.with(Serdes.UUID(),
eventAvroSerdes.createComplexEventRawSerde()));
        return complexEventsRaw;
    }

    @Bean
    public KStream<UUID, ComplexEventAvro>
buildComplexEventTopology(StreamsBuilder streamsBuilder) {

        //1. На вход подаются сырые комплексные события, которые
должны обогащаться метаданными
        var complexEventsRaw = streamsBuilder.stream(
                complexEventRawTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createComplexEventRawSerde())
        );
        //2. Создается состояние метаданных комплексных события
        var complexEventsMetadata = streamsBuilder.globalTable(
                complexEventMetadataTopic,
                Consumed.with(Serdes.UUID(),
eventAvroSerdes.createComplexEventMetadataSerde())
        );

        // 1. Обогащение комплексных событий
        // Сопоставление ключей на основании получаемого объекта
```

```
из соединения
        // [тип ключа потока основного потока|тип значения
основного потока|тип ключа просматриваемого потока ]
        KeyValueMapper<UUID, ComplexEventRawAvro, UUID> keyMapper
= (key, value) ->

UUID.fromString(value.getComplexEventMetadataId());
        // Соединитель между потоками сырого агрегированного
события и метаданными агрегированного события
        ValueJoiner<ComplexEventRawAvro, ComplexEventMetadataAvro,
ComplexEventAvro> complexEventJoiner =
                complexEventMapper::toEnriched;
        var aggregatedEvents = complexEventsRaw
                .join(complexEventsMetadata, keyMapper,
complexEventJoiner);

        //4. Отвлеченная бизнес логика(вывод на консоль)
        //TODO: добавить сервис
        aggregatedEvents.print(Printed.<UUID,
ComplexEventAvro>toSysOut().withLabel("complex-events"));

        //5. Далее записываем в топик
        aggregatedEvents.to(complexEventTopic,
Produced.with(Serdes.UUID(),
eventAvroSerdes.createComplexEventSerde()));

        return aggregatedEvents;
    }
```