Аннотация

Проектирование системы для потоковой обработки и агрегации данных социальных сетей

Кидун Станислав Русланович

Данное исследование направлено на разработку высокопроизводительной системы потоковой обработки данных социальных сетей в реальном времени. Предложена гибридная архитектура, интегрирующая принципы Каппа-архитектуры для устранения дублирования логики и микросервисный подход для обеспечения модульности и горизонтальной масштабируемости. Реализовано решение на базе Apache Kafka (шина событий с "ровно один раз"семантикой) и Apache Flink (движок потоковой обработки), гарантирующее задержку менее 60 секунд от публикации поста в VK до его обработки. Система протестирвоана на основе данных VK. Разработан механизм сбора данных через асинхронный фреймворк vkbottle с динамическим распределением нагрузки по API-токенам, обходящий ограничение на частоту запросов (3 запроса/секунду). Система рассматривает возможности реализации системы хранения данных: Elasticsearch для оперативной аналитики и полнотекстового поиска, Cassandra — для долгосрочного архивирования и интеграции с пакетной обработкой. Практическая проверка подтвердила линейную масштабируемость: обработка 500 сообществ требует 3 токена VK API, 10 000 сообществ — 50 токенов при сохранении целевой задержки. Разработанная платформа имеет практическую значимость для маркетингового анализа трендов, социологических исследований и кризисного мониторинга аномалий, демонстрируя эффективность гибридного подхода для задач Big Data и открывая пути для интеграции новых источников (Telegram, Twitter) и ML-моделей классификации контента.

Содержание

1 Введение

Начало текущего тысячелетия отличилось стремительным развитием цифровых технологий, которые проникли почти во все сферы человеческой деятельности. Одним из ключевых драйверов этого процесса стало повсеместное использование социальных сетей, генерирующих колоссальные объёмы данных в режиме реального времени. Эти данные содержат ценную информацию о поведении пользователей, их предпочтениях и социальных взаимодействиях, что делает их критически важными для бизнеса, маркетинга, социологических исследований и управления общественными процессами. Однако обработка таких данных сопряжена с рядом вызовов, включая необходимость обеспечения высокой производительности, масштабируемости и низкой задержки при анализе потоковой информации.

Существующие решения для работы с большими данными, такие как пакетная обработка, демонстрируют ограниченную эффективность в контексте динамичных социальных сетей, где задержки в анализе могут привести к утрате актуальности информации. В этой связи особую значимость приобретают системы потоковой обработки, способные агрегировать и анализировать данные в реальном времени. Подобные системы позволяют выявлять тенденции, обнаруживать аномалии и принимать оперативные решения на основе актуальных данных.

1.1 Определения используемые в данной работе

Введём несколько определений используемых в данной работе.

- Потоковая обработка данных — это процесс обработки и объединения данных в режиме реального времени по мере их поступления из непрерывных источников.
- Пакетная обработка данных — это процесс обработки, где данные накапливаются и анализируются постфактум.
- Бэкенд (Backend) — это внутренняя часть программной или информационной системы, которая отвечает за ее функциональность и обработку данных.
- Фронтенд (Frontend) в контексте веб-разработки относится к той части веб-приложения, которую видит пользователь и с которой он непосредственно взаимодействует
- СУБД — это программа, которая позволяет создавать базы данных и управлять ими. Она обеспечивает надёжное хранение информации, а также её безопасность и целостность. Помимо этого, СУБД предоставляет администратору средства для управления базой данных.
- API (Application Programming Interface) — это программный интерфейс, который позволяет одной компьютерной программе взаимодействовать с другими программами. Он предоставляет набор функций, объектов или действий, которые могут быть использованы разработчиками для создания приложений.
- IoT Мониторинг (IoT-monitoring, от Internet of Things) — процесс отслеживания, анализа и управления производительностью и здоровьем подключенных устройств в экосистеме Internet of Things. Он обеспечивает бесперебойную работу систем, повышает их эффективность и безопасность, предоставляя данные в режиме реального времени для дальнейшего анализа и упреждения проблем.

- JSON (расшифровывается как JavaScript Object Notation) — это облегченный формат для хранения и передачи данных. Он использует человекочитаемый текст для представления объектов данных, состоящих из пар <ИМЯ_СВОЙСТВА>: <ЗНАЧЕНИЕ> и массивов, что упрощает его понимание и использование в различных языках программирования.
- RPC (Remote Procedure Call) —
- RDD (Resilient Distributed Datasets)

1.2 Актуальность темы

В современных условиях социальные сети играют центральную роль в формировании информационной среды. Объёмы и скорость публикации данных в соцмедиа (Twitter, Facebook, VK и др.) растут крайне быстро: пользователи ежесекундно публикуют миллионы публикаций, фотографий и видео. Эти данные крайне ценны для самых разных областей — от маркетинга (анализ потребительских предпочтений и отслеживание репутации брендов) до общественной безопасности (мониторинг чрезвычайных ситуаций, выявление фейковых новостей или экстремистских проявлений). Однако эффективная работа с такими потоками требует специальных архитектур и технологий, способных обрабатывать данные в реальном времени с высокими объёмами и низкими задержками. Дело в том, что информация крайне быстро теряет актуальность, задержка в несколько минут или часов может сделать информацию устаревшей. В условиях постоянного роста объёмов данных традиционные методы пакетной обработки уже не отвечают требованиям времени. Отслеживание в потоковом режиме трендов и тенденций, ключевых событий или настроений аудитории даёт существенное преимущество в принятии решений в сфере маркетинга, в то время как своевременное обнаружение дезинформации может оказать значительное влияниие на общественную информацию. Поэтому возникает необходимость перехода к архитектурам, способным обеспечивать масштабируемость, отказоустойчивость и низкие задержки.

2 Постановка задачи

2.1 Цели работы

Цель этого исследования заключается в исследовании методов обработки и агрегации данных, разработке собственной системы, а так же поиска возможностей для оптимизации и улучшения производительности в парадигме работы с данными социальных сетей. В силу больших объёмов данных система должна быть способной обеспечить высокую производительность а так же обладать качествами высоконагруженных систем, таких как отказоустойчивость, высокую скорость отклика и быстрое восстановление после падения. Так же в силу разнообразности соцмедиа и отсутствия единых стандартов взаимодействия с ними, система должна обладать высокой гибкостью, что бы обеспечить возможность другим разработчикам создавать свои собственные модули без необхадимости перерабатывать уже разработанные модули. Так же одним из самых важных аспектов системы должно быть оперативное принятие решений на основе анализа поступающих данных. Итого, финальная модель должна соответствовать следующим критериям:

- **Масштабируемость.** Количество пользователей соцсетей продолжает расти, и система должна устойчиво работать при увеличении трафика в десятки раз без радикальной переработки инфраструктуры.
- **Отказоустойчивость.** Система должна уметь быстро восстанавливаться после падения, для скорейшего восстановления обработки трафика.
- **Гибкость разработки. Модульность.** В связи с большим разнообразием социальных сетей и возможным появлением новый в будущем, система должна иметь возможность интеграции модулей работы с ними без кординальной переработки.
- **Гибридность с офлайновой аналитикой.** Компании и исследовательские организации хотят сочетать быструю первичную аналитику с более глубокой офлайновой аналитикой на больших исторических объёмах данных. Правильное проектирование стриминговой системы должно учитывать интеграцию с хранилищами больших данных и системами batch-аналитики.
- **Низкая задержка и высокая скорость отклика.** В рамках социальных сетей данные могут быстро терять актуальность поэтому система должна быстро реагировать на новые данные.

2.2 Задачи работы

Для достижения поставленной цели были сформулированы следующие задачи:

- Изучить и проанализировать современные технологии и архитектуры потоковой обработки данных. Исследовать возможность их применения для работы с социальными сетями.
- Определить основные функциональные и нефункциональные требования к системе с учётом особенностей источников данных и существующих компонентов и решений.
- Разработать архитектурное решение, включающее выбор подходящих компонентов для обработки, агрегации и хранения данных.

- Реализовать прототип системы на основе выбранных технологий и провести его экспериментальную проверку.
- Проанализировать результаты экспериментов и оценить эффективность разработанного решения в соответствии с поставленными требованиями и критериями.
- Провести анализ написанной системы на возможности её улучшения и расширения в будущем.

3 Анализ технологий и требований

Идея потоковой обработки больших объёмов данный появилась десяток лет назад и активно развивается с каждым годом. Это мотивирует появление новых технологий, компонентов и архитектур. В этой главе мы рассмотрим самый распространённые и актуальные для исследования технологии.

3.1 Существующие архитектуры

С появлением задач потоковой обработки данных стали развиваться архитектуры, специализированные на решении данных задач. Далее рассматриваются самые популярные из них: Lambda-архитектура, Каппа-архитектура и Microservices + Event Sourcing:

3.1.1 Lambda-архитектура

Lambda-архитектура представляет собой гибридную модель проектирования системы обработки данных, направленную на преодоление фундаментального противоречия между низкой задержкой и высокой точностью. Данная парадигма достигает баланса за счёт декомпозиции вычислительного процесса на два независимых, но синхронизированных слоя: пакетный (batch) и скоростной (speed).

Ядро Lambda-архитектуры базируется на трёх инвариантах:

1. **Неизменяемость данных (Immutability)** — Все входящие события сохраняются в сыром виде в устойчивом хранилище без модификаций.

2. **Принцип пересчёта (Recomputation)** — Любое производное представление генерируется исключительно на основе первичных данных через детерминированные функции.

3. **Иерархия точности** — Пакетный слой формирует эталонные результаты на полном историческом срезе, тогда как скоростной слой компенсирует его инерционность приближёнными инкрементальными вычислениями.

Далее рассмотрим архитектурные компоненты и их взаимодействие:

- **Пакетный слой (Batch Layer)** функционирует как источник истины (source of truth). Он обрабатывает полный архив неизменяемых данных, используя распределённые вычислительные модели типа MapReduce. Результатом являются предварительно агрегированные представления (batch views), например, суточные статистики пользовательской активности. Эти представления материализуются в высокопроизводительных хранилищах типа колоночных баз данных. Ключевой характеристикой слоя является гарантированная точность, достигаемая за счёт полного пересчёта при изменении алгоритмов.

- **Скоростной слой (Speed Layer)** оперирует исключительно новыми событиями, ещё не учтёнными в batch views. Его задача — минимизировать задержку предоставления актуального состояния. Для этого применяются потоковые движки (Apache Storm, Flink), обрабатывающие данные в режиме реального времени. Результаты фиксируются в отдельном хранилище (real-time views), например, key-value базах. Слой жертвует точностью в пользу скорости: возможны временные расхождения из-за отсутствия исторического контекста.

- **Сервисный слой (Serving Layer)** обеспечивает согласованность данных для потребителей. При выполнении запроса он объединяет результаты batch views и real-time views через операцию слияния. Например, суммарное количество транзакций за текущий день будет вычислено как:

$$[result] = [speed\_statistics] + [batch\_statistics] \tag{1}$$

Несмотря на концептуальную стройность, Lambda-архитектура сталкивается с существенными проблемами в эксплуатации:

- **Семантический разрыв (Semantic Gap)** — Дупликация бизнес-логики в batch- и speed-слоях ведёт к рискам несогласованности. Например, оконная агрегация в Spark (batch) и Flink (stream) может давать различные результаты из-за различий в реализации.

- **Эксплуатационная сложность** — Поддержка двух независимых вычислительных кластеров (Hadoop для batch, Storm/Flink для stream) требует значительных ресурсов и координации. Синхронизация форматов данных, версий кода и метаданных становится нетривиальной задачей.

- **Латентность обновления эталонных данных** — Периодичность пересчёта batch views (например, раз в сутки) создаёт окна неактуальности, критичные для приложений реального времени.

- **Избыточность хранения** — Сырые данные дублируются в хранилищах для batch (HDFS) и stream (Kafka), увеличивая затраты.

Отраслевая практика показала, что классическая Lambda-архитектура эффективна лишь для узкого класса задач (например, финансовый аудит). В ответ на её недостатки возникли модификации:

- **Гибридные вычисления (Unified Processing)** — Использование фреймворков типа Apache Beam позволяет выполнять один код в режиме batch и stream, устраняя дублирование логики.

- **Оптимизация через Kappa-принципы** — Исторические данные воспроизводятся из потокового лога (Kafka с долгим хранением), сокращая зависимость от пакетного слоя.

- **Stateful Stream Processing** — Движки вроде Flink минимизируют разрыв между точностью и скоростью за счёт управляемых состояний (managed state) и чекпоинтов.

В заключении можно сказать, что Lambda-архитектура остаётся значимой теоретической моделью, иллюстрирующей онтологический конфликт между требованиями к задержке и точности в распределённых системах. Её главное наследие — явное разделение ответственности за консистентность (batch) и оперативность (speed). Однако операционные издержки и развитие потоковых технологий смещают фокус в сторону унифицированных подходов. Современные реализации заимствуют идею иерархии точности, но реализуют её через детерминированную реплей данных и транзакционные обновления состояний, что снижает необходимость в дуализме вычислений. Для новых систем целесообразна оценка компромиссов между чистотой Lambda и прагматикой Каппа-архитектур с учётом специфики предметной области.

3.2 Карра Архитектура

Каппа-архитектура возникла как методологический ответ на операционные сложности Lambda-подхода, предлагая принципиально иную философию проектирования. Её ядро составляет концепция единого вычислительного конвейера, где все данные — исторические и реального времени — обрабатываются через единую потоковую модель. Эта парадигма устраняет дуализм вычислений, характерный для Lambda, за счёт абстракции неизменяемого лога событий как универсального источника истины.

Каппа-архитектура базируется на трёх фундаментальных принципах:

1. **Централизация на потоковом логе** — Все входящие события бессрочно сохраняются в распределённом брокере сообщений (Apache Kafka, Pulsar) с гарантией сохранности и порядковой целостности.

2. **Обработка через реплей** — Для пересчёта результатов при изменении логики или восстановлении после сбоев система заново обрабатывает релевантные сегменты лога.

3. **Детерминизм вычислений** — Повторный прогон тех же данных через обновлённую логику обязан давать консистентный результат, что обеспечивается неизменяемостью событий и идемпотентностью операторов.

Рассматривая Каппа-Архитектуру более подробно можно выделить следующие структурные компоненты:

1. Неизменяемое хранилище событий выполняет роль первичного источника данных. Брокер сообщений организует события в упорядоченные, секционированные последовательности (топики), где каждая запись снабжается монотонно растущим смещением (offset). Долговременное хранение (ретеншн) обеспечивает доступ к полному историческому архиву — от часов до лет в зависимости от требований.

2. Единый вычислительный слой заменяет дублирующие batch- и speed-компоненты Lambda. Потоковый движок (Apache Flink, Spark Structured Streaming) потребляет данные непосредственно из лога, применяя пользовательскую логику. Ключевая инновация — поддержка репроцессинга: при необходимости система может перезапустить обработку с произвольной точки смещения, игнорируя текущее состояние. Для управления таким пересчётом используются:

   - **Точки восстановления (savepoints)** — снапшоты состояния вычислений;
   - **Водяные знаки (watermarks)** — механизмы контроля полноты данных при обработке по времени событий.

3. Сервисный интерфейс предоставляет результаты через материализованные представления. В отличие от Lambda, здесь отсутствует операция слияния — актуальное состояние генерируется исключительно потоковым конвейером и сохраняется в OLAP-хранилищах (ClickHouse), key-value базах (Redis) или специализированных движках (Apache Pinot).

Говоря про математическую модель архитектуру стоит выделить процесс обработки данных. Выглядит он следующим образом: Результат вычислений R в момент t определяется как функция от полного набора событий $S_{[0,t]}$:

$$R(t) = \Phi(S_{[0,t]}, \Omega) \tag{2}$$

где:

- Φ — потоковая функция обработки,
- Ω — конфигурация состояния (окна, агрегаторы, UDF).

При изменении Ω (например, исправлении алгоритма) результат пересчитывается как

$$R`(t) = \Phi_{new}(S_{[0,t]},\Omega_{new}). \quad (3)$$

Несмотря на элегантность подхода, архитектура предъявляет жёсткие требования к инфраструктуре:

- **Производительность брокера** — Длительный ретеншн требует экстремальной масштабируемости хранилища. Например, топик Kafka с ретеншном 1 год при нагрузке 1 ТВ/день потребует 365 ТВ дискового пространства с учётом репликации.
- **Детерминизм операций** — Некорректная обработка времени событий (временных штампов) или неидемпотентные операции приведут к расхождениям при реплее.
- **Ресурсоёмкость репроцессинга** — Полный пересчёт истории может занимать часы, блокируя ресурсы кластера. Техники инкрементального обновления состояний (например, в Flink) смягчают, но не решают проблему.
- **Сложность отладки** — Анализ причин расхождений между разными версиями $R()$ и $R`()$ требует трассировки событий по всему конвейеру.

В сравнении с Lambda-архитектурой, Каппа-архитектура устраняет три фундаментальных ограничения:

1. **Ликвидация дублирования логики** — Единый код для всех данных вместо раздельных реализаций под batch/stream.
2. **Упрощение инфраструктуры** — Отказ от пакетного кластера (Hadoop/Spark) в пользу единого потокового стека.
3. **Гарантия согласованности** — Отсутствие семантического разрыва между "точными"и "приближёнными"представлениями.

Однако Каппа менее устойчива к ошибкам в проектировании: некорректная водяная метка или потеря состояния при реплее фатальны. Lambda же изолирует риски в speed-слое, сохраняя эталонные данные в batch.

Итого, Каппа-архитектура представляет собой эволюционный шаг в направлении унификации вычислительных моделей. Её главное достижение — доказательство возможности построения консистентных систем реального времени без дуализма вычислений. Хотя операционные требования остаются высокими, развитие управляемых сервисов (Confluent Cloud, Amazon MSK) и стандартизация семантик "ровно один раз"(Kafka Transactions, Flink Checkpoints) снижают барьер внедрения. Для современных разработок Каппа служит предпочтительной основой при условии: долговременной сохраняемости событий в брокере, детерминированности бизнес-логики и готовности к компромиссам в ресурсоёмкости репроцессинга. В контексте научной дискуссии она демонстрирует сдвиг парадигмы: от компромиссов между точностью и скоростью — к их синтезу через иммутабельные данные и реиграцию состояний.

3.2.1 Microservices + Event Sourcing

Микросервисная архитектура, интегрированная с принципами проектирования, опирающихся на события (Event Sourcing), формирует парадигму, где потоки событий становятся центральной осью координации и сохранения состояния. Данный подход трансформирует традиционные модели взаимодействия сервисов через API в декомпозированную систему асинхронной коммуникации, основанной на неизменяемых фактах.

Философия архитектуры базируется на двух взаимодополняющих принципах:

1. **Проектирование с упором на события (Event Sourcing)** — Состояние системы выводится исключительно из последовательности событий, фиксирующих изменения. В отличие от CRUD-моделей, где актуальное состояние хранится напрямую, здесь реконструкция объекта осуществляется путём применения событий в порядке их возникновения.

2. **Декомпозиция на микросервисы** — Каждый сервис инкапсулирует собственную бизнес-логику и данные, взаимодействуя с другими исключительно через события. Это устраняет жесткие зависимости, характерные для монолитных систем.

Ключевым объектом подобного подхода являются брокеры сообщений (Apache Kafka, RabbitMQ), которые одновременно выступают потоками, связывающими сервисы и журналом команд, гарантирующим сохранность и порядок событий.

Механизмы взаимодействия компонентов:
Генерация событий инициируется командами, поступающими в сервисы через общий шлюз, часто называемый Gateway, или от других сервисов. Например, сервис обработки заказов при подтверждении платежа генерирует событие OrderPaid, которое публикуется в соответствующий топик. Каждое событие обычно содержит:

- Уникальный идентификатор агрегата (например, order_id)
- Таймстамп генерации
- Полезную нагрузку в формате, независимом от потребителей

Потребители событий (микросервисы-консьюмеры) подписываются на релевантные топики, обновляя свои локальные состояния или инициируя новые процессы. Сервис аналитики может использовать OrderPaid для расчёта метрик выручки, а сервис доставки — для планирования маршрутов.

Однако данный подход имеет ряд вызовов и проблем, которые усложняют работу:

- **Сложность отслеживания причинности** — В распределённых системах события могут поступать с задержками, нарушая локальный порядок. Механизмы водяных знаков (watermarks) и векторных часов частично решают проблему.
- **Управление схемами данных** — Эволюция форматов событий требует стратегий обратной совместимости (Schema Registry, Avro).
- **Ресурсоёмкость пересчёта состояния** — Длительные цепочки событий замедляют восстановление сервисов после сбоев. Оптимизация через снапшоты (snapshots) текущих состояний.
- **Отладка распределённых процессов** — Трассировка транзакций, затрагивающих несколько сервисов, требует инструментов вроде OpenTelemetry.

- **Гарантии доставки** — Хотя брокеры типа Kafka обеспечивают "ровно один раз"семантику, идемпотентность обработчиков остаётся ответственностью разработчика.

Паттерн CQRS (Command Query Responsibility Segregation) дополняет модель и решает часть проблем, разделяя операции записи и чтения. Синхронизация между частями в таком подходе обеспечивается асинхронно через поток событий, что устраняет блокировки при высокой нагрузке. В таком подходе модель разделяется на следующие компоненты:

1. **Командная сторона (Write Model)** — обрабатывает команды, валидирует их и генерирует события.
2. **Сторона запросов (Read Model)** — строит материализованные представления из потока событий (например, в Elasticsearch для полнотекстового поиска).

В сравнении с другими архитектурами, микросервисных подход в связке с проектированием, опирающимся на события, имеет следующие приемущества:

- **Аудит и воспроизводимость** — Полная история событий позволяет реконструировать состояние системы на любой момент времени.
- **Отказоустойчивость** — Потеря состояния сервиса компенсируется переигранием событий из журнала.
- **Гибкая эволюция** — Добавление новых сервисов не требует изменения существующих: достаточно подписки на релевантные события.
- **Масштабируемость** — Независимое шардирование топиков и консьюмерских групп распределяет нагрузку.
- **Согласованность** — Асинхронная модель гарантирует консистентность без распределённых транзакций.

3.3 Системы обработки событий

Одними из ключевых технологий в задаче потоковой обработки данных являются системы обработки событий. Они выступают в качестве главных артерий системы и решают задачу общения между различными компонентами систимы, а так же занимаются вопросом перемещения больших объёмов данных. Такие системы чаще всего работают с понятиями "потока"или "очереди которые является основной сущностью, используемой для передачи данных. Вот самые распространённые из них:

3.3.1 Apache Kafka

Apache Kafka представляет собой распределённую платформу для потоковой передачи и обработки данных, разработанную для обеспечения высокой пропускной способности, минимальных задержек и устойчивости к сбоям. Её архитектура основана на модели публикации-подписки, что позволяет эффективно управлять непрерывными потоками информации между разнородными источниками и приложениями. Ключевой особенностью системы является способность обрабатывать миллионы событий в секунду, сохраняя при этом целостность и порядок данных даже в условиях масштабирования или частичных отказов узлов.

Основу Kafka составляют топики — логические каналы, через которые данные распределяются между участниками системы. Каждый топик делится на партиции, обеспечивающие параллелизм обработки: записи в рамках партиции упорядочены, что гарантирует консистентность, а распределение партиций между узлами кластера позволяет балансировать нагрузку. Продюсеры — компоненты, ответственные за публикацию данных в топики — взаимодействуют с брокерами (серверами Kafka), которые сохраняют записи на диск с заданной политикой хранения. Консьюмеры, в свою очередь, извлекают данные из топиков, поддерживая как индивидуальное, так и групповое потребление, что обеспечивает гибкость в распределении задач.

Важным аспектом архитектуры Kafka является её отказоустойчивость. Механизм репликации позволяет создавать копии партиций на нескольких брокерах, гарантируя доступность данных при выходе узлов из строя. Для координации метаданных кластера (например, информации о расположении партиций) ранее использовался Apache ZooKeeper, однако современные версии Kafka постепенно внедряют встроенные решения для уменьшения внешних зависимостей.

Система поддерживает несколько семантик доставки сообщений:

- **"максимум один раз"** — данные могут быть потеряны, но не доставлены повторно;
- **"минимум один раз"** — гарантируется минимум одна доставка, что может приводить к дублированию;
- **"ровно один раз"** — исключает как потери, так и повторения, достигая идемпотентности операций.

3.3.2 Apache Flink

Apache Flink представляет собой высокопроизводительный фреймворк с открытым исходным кодом, ориентированный на обработку потоковых и пакетных данных в гибридном формате. В отличие от систем, разделяющих подходы к работе с потоками и статическими наборами данных, Flink унифицирует эти модели через абстракцию бесконечных потоков, где пакетная обработка рассматривается как частный случай потоковой с фиксированным временным окном. Это позволяет разработчикам создавать гибридные приложения, сочетающие аналитику в реальном времени и исторические вычисления.

Ключевой особенностью архитектуры Flink является управление состоянием (state management). Система сохраняет промежуточные результаты обработки событий, обеспечивая консистентность даже при сбоях узлов. Механизм чекпоинтов (checkpoints) периодически фиксирует состояние всех операторов в распределённом хранилище, что гарантирует восстановление данных с точностью до последнего корректного состояния. Для достижения семантики "ровно один раз"Flink использует двухфазный протокол фиксации транзакций, что исключает дублирование или потерю информации — критически важное свойство для финансовых систем или аудита.

Одним из преимуществ Flink перед аналогами (например, Apache Spark) является нативная поддержка потоковой обработки. Вместо имитации потоков через микропакеты Flink оперирует непрерывными данными, что снижает задержки и повышает эффективность ресурсов. Архитектура фреймворка включает TaskManager (исполнительные узлы) и JobManager (оркестратор задач), распределяющий операции по кластеру. Для балансировки нагрузки Flink динамически перераспределяет задачи между узлами, используя модель data parallelism.

Важным аспектом Flink является работа с временными метками. Система различает:

- **Event time (временной штамп)** — время генерации события источником
- **Processing time** — момент обработки события фреймворком
- **Ingestion time** — время поступления данных в Flink. Поддержка временных штампов позволяет корректно обрабатывать задержанные или неупорядоченные данные, что актуально для IoT-устройств или глобальных систем с неравномерной задержкой сети.

Flink предоставляет богатый набор операторов для трансформации данных:

- **Оконные функции (tumbling, sliding, session windows)** — группировка событий по временным или количественным критериям;
- **CEP (Complex Event Processing)** — обнаружение паттернов в потоках (например, последовательность действий пользователя);
- **Интеграция с ML-библиотеками** — выполнение прогнозных моделей в реальном времени.

3.3.3 RabbitMQ

RabbitMQ представляет собой брокер сообщений с открытым исходным кодом, реализующий протокол AMQP (Advanced Message Queuing Protocol). Его основная задача — обеспечить надёжную асинхронную коммуникацию между компонентами распределённых систем, что особенно актуально в микросервисных архитектурах и сценариях, требующих декомпозиции задач. В отличие от систем, ориентированных на потоковую обработку, RabbitMQ фокусируется на гарантированной доставке сообщений, управлении очередями и балансировке нагрузки между потребителями.

Архитектурные компоненты системы включают:

- **Обменники (Exchanges)** — точки входа для сообщений, определяющие правила маршрутизации.
- **Очереди (Queues)** — буферы для хранения сообщений до их обработки потребителями.
- **Привязки (Bindings)** — правила, связывающие обменники с очередями на основе ключей маршрутизации.

Сообщения публикуются продюсерами в обменники, которые, используя заданный тип (direct, topic, fanout, headers), определяют, в какие очереди они будут направлены. Например, обменник типа fanout рассылает сообщения во все привязанные очереди, а topic позволяет использовать шаблоны для гибкой фильтрации.

Гарантии доставки обеспечиваются механизмами подтверждений (acknowledgments). Потребитель отправляет брокеру подтверждение после успешной обработки сообщения. Если подтверждение не получено (например, из-за сбоя), RabbitMQ повторно ставит сообщение в очередь. Для критически важных задач поддерживается режим persistent messages, при котором данные сохраняются на диск, предотвращая потерю при перезапуске сервера.

Кластеризация повышает отказоустойчивость системы. RabbitMQ позволяет объединять узлы в кластеры с общими метаданными (информацией об обменниках, очередях), но данные очередей хранятся локально на каждом узле. Для обеспечения избыточности используется зеркалирование очередей (mirrored queues), при котором сообщения реплицируются между узлами. Это минимизирует риск потери данных при выходе из строя отдельных серверов.

Примеры применения RabbitMQ охватывают широкий спектр задач:

- **Асинхронная обработка задач** — вынесение ресурсоёмких операций (генерация отчётов, обработка изображений) в фоновый режим.
- **Балансировка нагрузки** — распределение запросов между несколькими экземплярами сервиса через конкурентных потребителей.
- **Интеграция разнородных систем** — обеспечение взаимодействия между legacy-приложениями и современными микросервисами через унифицированный интерфейс обмена сообщениями.

В отличие от систем, оптимизированных для потоковой аналитики (например, Apache Kafka), RabbitMQ не предназначен для обработки экстремально высоких объёмов данных в реальном времени. Однако его сила заключается в точном управлении жизненным циклом сообщений, гибкой маршрутизации и поддержке сложных сценариев взаимодействия, таких как RPC или распределённые транзакции.

3.3.4 Apache Spark Streaming

Apache Spark Streaming представляет собой модуль фреймворка Apache Spark, предназначенный для обработки потоков данных через микропакетную модель (micro-batching). В отличие от систем, ориентированных на непрерывную потоковую обработку (например, Flink), Spark Streaming делит входящий поток на небольшие пакеты фиксированного интервала (от миллисекунд до секунд), обрабатывая их как статические RDD-наборы. Это позволяет использовать единую кодовую базу и API для пакетной и потоковой аналитики, обеспечивая согласованность в гибридных сценариях.

Он состоит из следующих архитектурных компонент:

- **Драйвер** — Оркестрирует задачи, разбивает поток на микропакеты (DStreams) и распределяет их по исполнителям.
- **Исполнители** — Обрабатывают пакеты данных на рабочих узлах кластера.
- **Высокоуровневый API** — Предоставляет абстракцию "бесконечной таблицы" над потоком, позволяя применять SQL-подобные операции и декларативные запросы.

Ключевыми особенностями можно выделить:

- **Гарантии "ровно один раз"** — Достигается через комбинацию чекпоинтов состояния (через HDFS/S3) и атомарной записи результатов (начиная с Structured Streaming).
- **Интеграция с экосистемой Spark** — Поддержка MLlib (машинное обучение в реальном времени), GraphX (графовая обработка) и Spark SQL.
- **Обработка по времени событий** — Возможность агрегации данных по временным окнам с учетом меток генерации события, а не времени поступления.

- **Восстановление через чекпоинты** — Восстановление состояния приложения после сбоя из контрольных точек.

В отличие от Apache Flink, использующего нативную потоковую модель, Spark Streaming жертвует сверхнизкой задержкой (типично 0.5–2 секунды) в пользу интеграции с пакетной обработкой Spark и богатой экосистемой. Для сценариев, требующих строгого порядка событий и латенции в миллисекунды, предпочтительнее Flink; для аналитических задач с умеренной задержкой и унифицированной кодобазой — Spark Streaming.

3.3.5 Apache Storm

Apache Storm представляет собой распределённый, отказоустойчивый движок реального времени (true streaming) для обработки неограниченных потоков данных с экстремально низкой задержкой (миллисекунды). Разработанный в Twitter для анализа твитов в реальном времени, Storm фокусируется на простой, высокопроизводительной доставке и трансформации событий, где порядок обработки каждого сообщения критичен. Его архитектура основана на направленном ациклическом графе (DAG) задач, называемом топологией (topology), которая определяет поток данных и логику обработки.

Архитектурные компоненты:

- **Nimbus** — Центральный координатор (аналог JobManager в Flink), отвечающий за распределение кода топологии по рабочим узлам, планирование задач и мониторинг сбоев. Является "главным"узлом кластера (но без единой точки отказа при использовании HA-режима).

- **Supervisor** — Демон, работающий на каждом рабочем узле кластера. Получает задания от Nimbus, запускает и останавливает рабочие процессы (Worker processes), выделенные под выполнение конкретных задач топологии.

- **Worker process** — JVM-процесс, выполняющий часть топологии. Запускает внутри себя Executor'ы (потоки), которые выполняют Task'и (экземпляры компонентов spout или bolt).

- **Spout** — Источник данных в топологии. Читает данные из внешних систем (Kafka, Kinesis, очереди, сокеты) и испускает кортежи (tuples) в топологию. Отвечает за надёжность: отслеживает обработку испущенных кортежей и повторяет отправку при сбоях (в режимах с гарантиями).

- **Bolt** — Обработчик данных. Принимает кортежи от spout'ов или других bolt'ов, выполняет над ними операции (фильтрация, агрегация, обогащение, запись в БД), и может испускать новые кортежи дальше по топологии. Bolt'ы образуют вычислительные вершины графа.

Одной из ключевых особенностей Apache Storm является модель True Streaming обработка событий по одному сразу при их поступлении, в отличие от микропакетных моделей (Spark Streaming). Это обеспечивает минимально возможную задержку. Так же отдельно стоит упомянуть возможноть для горизонтального масштабирования системы: Она легко масштабируется добавлением новых рабочих узлов. Параллелизм spout/bolt настраивается независимо.

Apache Storm так же может похвастаться отказоустойчивостью: При падении рабочего узла (Worker) Nimbus перезапускает его задачи на других узлах. При падении

самого Nimbus, работающие Supervisor'ы и Worker'ы продолжают функционировать до его восстановления (режим HA требует резервного Nimbus и ZooKeeper).

Apache Storm так же поддерживает те же гарантии доставки сообщений, что и Apache Kafka:

- **"максимум один раз"(По умолчанию)** — Кортежи могут быть потеряны при сбое, но не обработаны повторно (производительность выше).
- **"минимум один раз"** — Гарантирует, что каждый кортеж будет обработан минимум один раз (возможны дубли). Достигается механизмом "acknowledgment"(ACK): spout отслеживает кортежи и переиспускает не подтверждённые в течение таймаута.
- **"ровно один раз"(через Trident)** — Предоставляется высокоуровневой абстракцией Trident. Trident группирует кортежи в мелкие пакеты (микробатчи), использует уникальные ID для отслеживания и обеспечивает идемпотентность операций или транзакционное обновление состояния. Однако это увеличивает задержку и сложность.

Для Apache Storm можно выделить следующие отличия от аналогов:

1. **Apache Kafka** — Storm является кластерным фреймворком (требует Nimbus/Supervisors или YARN/K8s), тогда как Kafka — это легковесная система, интегрируемая почти в любое приложение и используемая для координации и хранения состояния. Kafka Streams проще для Kafka-центричных пайплайнов, но Storm универсальнее для источников, не связанных с Kafka.
2. **Apache Flink** — Storm имеет более слабую поддержку управления состоянием (state management) и обработки по времени события (временных штампов) по сравнению с Flink. Trident добавляет состояние и "ровно один раз"семантику, но за счет задержки и усложнения модели программирования. Flink изначально создан для управления состоянием и работы с штампами времени событий. Storm имеет меньшую абстрактную задержку, но Flink часто превосходит его в пропускной способности на сложных состояниях.
3. **Apache Spark Streaming** — Storm обеспечивает на порядок меньшую задержку (миллисекунды против секунд), так как обрабатывает события по одному, а не микропакетами. Однако Spark Streaming проще интегрируется с пакетной обработкой Spark и экосистемой (MLlib, SQL) и предоставляет более сильные гарантии состояния и временных штампов "из коробки"через Structured Streaming.

Storm остаётся эффективным выбором для задач, требующих минимально возможной задержки реакции на событие (например, мониторинг сетевой безопасности, простые фильтрации/обогащения в реальном времени, триггеры мгновенных реакций) и где сложное состояние или обработка по временным штампам не являются основным требованием. Его относительная простота также может быть преимуществом для конкретных сценариев. Однако для сложной аналитики с окнами, точным временем событий и богатым состоянием Flink или Spark Structured Streaming часто предпочтительнее.

4 Определение требований и компонентов

На основе проведённого анализа существущих технологий и архитектур, к системе были вынесены следующие функциональные и нефункциональные требования:

4.1 Требования к системе

4.1.1 Функциональные требования

Система должна обеспечивать выполнение следующих функциональных задач:

1. Система обеспечивает непрерывный прием данных, генерируемых пользователями. В качестве источника используются сообщества социальной сети ВКонтанте (VK)

2. Система обеспечивает обработку каждого поста (сообщения) с задержкой, не превышающей установленный порог (менее 60 секунд) от момента публикации в VK до завершения его обработки в рамках основного конвейера.

3. Система обеспечивает асинхронную, надежную и упорядоченную (в рамках партиций) передачу данных между микросервисами, ответственными за различные этапы обработки. Маршрутизация осуществляется через систему обработки событий, выступающую в качестве шины событий. Планируется использование Apache Kafka.

4. Система обеспечивает сохранение результатов обработки каждого поста в промежуточном хранилище с возможностью последующего извлечения для дальнейшего анализа и агрегации. Данные сохраняются в структурированном виде, пригодном для эффективного выполнения аналитических запросов.

5. Система обеспечит эффективный полнотекстовый поиск и сложный аналитический запрос по историческим данным, накопленным в результате работы. Планируется использование Elasticsearch для поиска и, возможно, Cassandra (или аналогичной распределенной NoSQL БД) для хранения детализированных данных, оптимизированных под высокую скорость записи и масштабируемость.

4.1.2 Нефункциональные требования

Система должна обладать следующими качественными характеристиками:

1. Система гарантирует обработку более 90% поступающих постов с задержкой менее 60 секунд от момента появления поста в VK до завершения его базовой обработки (FR2). Это требование является основным критерием "реального времени"в рамках данной работы. Система также должна обеспечивать пропускную способность, достаточную для обработки пиковой нагрузки, характерной для выбранных для мониторинга популярных пабликов VK.

2. Система проектируется с расчетом на горизонтальную масштабируемость. Увеличение объема обрабатываемых данных (числа отслеживаемых пабликов/постов) должно достигаться преимущественно за счет добавления вычислительных ресурсов (нод) и параллельных экземпляров микросервисов без изменения архитектуры. Компаненты системы должны уметь масштабироваться независимо друг от друга.

3. Система проектируется с расчётом на удобство интеграций сторонних сервисов и добавления нового функционала. Нарпимер добавление новой социальной сети к обработке, нового типа данных, внедрение машинного обучения или добавление новых сообществ и целей к обработке.

4.2 Выбор компонентов

4.2.1 Выбор языка программирования

При проектировании системы потоковой обработки данных социальных сетей одним из аспектов является выбор языка программирования, способного обеспечить баланс между производительностью, гибкостью интеграции и скоростью разработки. В качестве потенциальных кандидатов рассматриваются языки с различными парадигмами, включая C++, Java, Go и Python. Каждый из них обладает уникальными характеристиками, однако окончательный выбор должен учитывать специфику задач, связанных с агрегацией данных, взаимодействием с внешними API (например, VK API) и интеграцией с распределёнными системами (Apache Kafka, Flink). Исходя из этих критериев был выбран Python.

Критерии выбора

1. **Интеграционная гибкость** — поддержка библиотек для работы с разнородными сервисами и протоколами.
2. **Скорость разработки** — лаконичность синтаксиса и доступность инструментов для быстрого прототипирования.
3. **Производительность** — эффективность выполнения ресурсоёмких операций, таких как обработка потоков данных.
4. **Экосистема** — наличие готовых решений для машинного обучения, многопоточности, визуализации, интеграции распределённых систем и работы с Big Data.

Рассмотрим альтернативы и варианты. В качестве возможных языков были выбраны следующие: **C++**, **Java**, **Go**, **Python**. Проанализируем их сильные и слабые стороны с учётом заданных критериев.

Анализ альтернатив

1. **C++** демонстрирует высокую производительность за счёт низкоуровневого управления памятью и оптимизированной компиляции. Это делает его предпочтительным для задач, требующих минимальных задержек. Однако для интеграции с современными облачными сервисами и RESTful API необходима разработка дополнительных обёрток, что увеличивает сроки реализации. Кроме того, отсутствие встроенной поддержки асинхронных операций усложняет работу с потоковыми данными.
2. **Java** предлагает развитую экосистему (Spring, Apache Camel) и высокую портируемость за счёт JVM. Тем не менее, строгая статическая типизация и объёмный boilerplate-код замедляют итеративную разработку, что критично для динамично меняющихся требований социальных сетей.
3. **Go** выделяется простотой создания многопоточных приложений, но ограниченная поддержка библиотек для интеграции с NLP-фреймворками или аналитическими инструментами снижает его применимость в данном контексте.

4. **Python** предлагает высокую скорость разработки за счёт простоты языка и наличия большого числа готовых решений из различных отраслей. Так же Python предлагает широкий выбор специализированных библиотек, для интеграции разнородных сервисов.

Обоснование выбора: Python

Выбор Python в качестве основного языка для реализации платформы обусловлен необходимостью быстрой интеграции с разнородными сервисами, доступностью специализированных библиотек и снижением временных затрат на разработку. Несмотря на компромиссы в производительности, его экосистема предоставляет инструменты для построения масштабируемых и maintainable-решений, соответствующих требованиям современных социальных медиа.

Python, несмотря на более низкую производительность в сравнении с компилируемыми языками, становится оптимальным решением благодаря следующим факторам:

1. **Богатая экосистема библиотек:**
   - **Интеграция с API** — Библиотеки requests, aiohttp, vkbottle и vk-api упрощают взаимодействие с социальными сетями (VK, Telegram) и внешними сервисами.
   - **Работа с потоковыми данными** — Kafka-python, pyflink и faust обеспечивают совместимость с Apache Kafka и Flink, позволяя реализовывать сложные ETL-процессы.
   - **Аналитика и ML** — Pandas, NumPy, scikit-learn и TensorFlow поддерживают обработку данных и внедрение моделей машинного обучения. Это может быть полезно при дальнейшем разширении и развитии.
2. **Асинхронная обработка:**
   Модуль asyncio и фреймворки (FastAPI, Quart) позволяют эффективно управлять тысячами одновременных подключений, что критично для систем агрегации данных в реальном времени.
3. **Скорость разработки:**
   Динамическая типизация и лаконичный синтаксис сокращают время написания кода. Простота эксплуатации и запуска значительно упрощают тестирование гипотез.
4. **Сообщество и документация:**
   Широкая популярность Python гарантирует доступ к актуальным руководствам, форумам и open-source проектам, что минимизирует риски возникновения неразрешимых проблем.
5. **Гибкость масштабирования:**
   Для оптимизации производительности критических участков кода возможно использование C-расширений (Cython) или интеграция с высокопроизводительными фреймворками (Dask, Ray).

4.2.2 Интеграция с VK API

Интеграция с API социальной сети ВКонтакте (VK) является критически важным компонентом платформы, обеспечивающим сбор данных, управление контентом и взаимодействие с пользователями. При выборе библиотеки для работы с VK API необходимо

учитывать такие факторы, как поддержка асинхронных операций, удобство работы с методами API, качество документации и совместимость с архитектурой системы. Среди доступных решений (например, vk_api, aiovk, vkbottle) анализ функциональности и производительности позволил определить оптимальный вариант — фреймворк vkbottle.

Критерии выбора

1. **Асинхронная обработка запросов** — минимизация задержек при работе с сетевыми операциями.
2. **Поддержка Long Poll API** — эффективный механизм для получения событий в реальном времени из источников, к которым имеются права администрации.
3. **Гибкость архитектуры** — возможность кастомизации обработчиков и middleware.
4. **Совместимость с экосистемой Python** — интеграция с asyncio и другими библиотеками.

Рассмотрим альтернативы и варианты. В качестве возможных библиотек были выбраны следующие: **vk_api**, **vkbottle**, **aiovk**. Проанализируем их сильные и слабые стороны с учётом заданных критериев.

Анализ альтернатив

1. **vk_api** — Широко используемая библиотека, предоставляющая базовый функционал для работы с API. Однако отсутствие нативной поддержки асинхронности делает её непригодной для высоконагруженных систем, где параллельная обработка тысяч запросов является обязательным требованием.
2. **vkbottle** — Распространённая всинхронная библиотека. Имеет широкий функционал и подробную документацию. Поддерживает кастомизацию процессов, для более тонкой настройки.
3. **aiovk** — Библиотека с асинхронным подходом, основанная на aiohttp. Несмотря на улучшенную производительность, её функционал ограничен базовыми методами API, а документация недостаточно детализирована для сложных сценариев.

Обоснование выбора vkbottle

Выбор vkbottle в качестве основы для взаимодействия с VK API обусловлен его способностью эффективно решать задачи асинхронной обработки событий, сохраняя высокую производительность в условиях интенсивной нагрузки. Это согласуется с общей архитектурой платформы, где ключевыми требованиями являются масштабируемость, минимальные задержки и интеграция с распределёнными системами. Использование данной библиотеки не только оптимизирует работу с API, но и обеспечивает основу для будущего расширения функционала, такого как внедрение чат-ботов или автоматизированных служб поддержки. Его архитектура предоставляет следующие преимущества:

1. **Асинхронная модель выполнения** — Использование asyncio позволяет обрабатывать множество запросов параллельно без блокировки основного потока. Это особенно важно для социальных сетей, где задержки в получении данных (например, комментариев или лайков) могут привести к потере актуальности информации. Благодаря неблокирующим операциям, vkbottle эффективно использует ресурсы, что нивелирует ограничения Python, связанные с глобальной блокировкой интерпретатора (GIL).

2. **Модульность и расширяемость** — vkbottle поддерживает множественные запросы, в том числе с использованием более чем одного ключа, что позволяет оптимизировать процессы. Так же vkbottle поддерживает создание кастомных middleware, что позволяет внедрять дополнительную логику (например, логирование, кеширование, валидацию) на разных этапах обработки запросов. Интеграция с DI-контейнерами (Dependency Injection) упрощает управление зависимостями в крупных проектах.

3. **Совместимость с аналитическими инструментами** — Библиотека легко интегрируется с другими . Это обеспечивает беспрепятственное взаимодействие с Apache Kafka (через kafka-python) и Apache Flink, что критично для сквозной обработки данных.

Использование vkbottle позволяет избежать узких мест, характерных для синхронных подходов. Например, при одновременной обработке входящих сообщений от тысяч пользователей асинхронная модель обеспечивает линейное масштабирование производительности без увеличения задержек. Кроме того, фреймворк поддерживает автоматическое повторение неудачных запросов и обработку квот API, что повышает отказоустойчивость системы.

4.2.3 Выбор системы обработки событий

Для реализации потоковой обработки данных социальных сетей критически важен выбор технологической платформы, обеспечивающей масштабируемость, отказоустойчивость и низкую задержку. Рассмотрим основные системы, релевантные для решения поставленной задачи, и выделим из анализа, проведённого в предыдущей главе, все ключевые особенности.

Критерии выбора

1. **Гарантии доставки** — обеспечение семантики "ровно один раз"для исключения дублирования и потерь данных при сбоях, что принципиально важно для точности аналитических отчётов.

2. **Задержка обработки** — максимально допустимое время между поступлением события и получением результата его обработки, определяющее возможность работы в режиме реального времени.

3. **Пропускная способность** — способность системы обрабатывать пиковые нагрузки, характерные для социальных сетей, измеряемая в событиях/секунду при сохранении стабильной задержки.

4. **Управление состоянием** — возможности эффективного хранения и восстановления контекста обработки (агрегаторов, сессий пользователей) при горизонтальном масштабировании и отказах узлов.

5. **Горизонтальная масштабируемость** — автоматическое распределение нагрузки между узлами, перебалансировка партиций без прерывания обслуживания и обработка асимметрии нагрузки.

6. **Операционная устойчивость** — встроенные механизмы восстановления после сбоев, мониторинг сквозной задержки (end-to-end latency) и интеграция с системами алертинга.

Apache Kafka

Apache Kafka представляет собой распределённую платформу для потоковой передачи данных, функционирующую как высокопроизводительный брокер сообщений. Её архитектура основана на модели публикации-подписки с разделением данных на тематические каналы (топики), каждый из которых делится на партиции для обеспечения параллелизма. Уникальной особенностью Kafka является комбинация характеристик:

- Гарантированная пропускная способность на уровне миллионов сообщений в секунду
- Минимальные задержки доставки (менее 10 мс)
- Механизм репликации данных для отказоустойчивости
- Долговременное хранение сообщений с настраиваемой политикой

Важным аспектом является поддержка семантики *"ровно один раз"* через транзакционные продюсеры, что исключает дублирование или потерю данных. Для аналитических задач Kafka предоставляет библиотеку Kafka Streams и движок ksqlDB, позволяющий выполнять SQL-запросы над потоками в реальном времени.

Apache Flink

Apache Flink является фреймворком для распределённой обработки потоковых и пакетных данных с акцентом на низкую задержку и точные семантики. Его архитектурное отличие от аналогов заключается в нативной реализации потоковой модели (true streaming), где данные обрабатываются по мере поступления, а не разбиваются на микропакеты. Ключевые особенности включают:

- Поддержка обработки по времени событий (event-time processing)
- Распределённые снапшоты состояния (distributed snapshots)
- Точную семантику *"ровно один раз"* без снижения производительности
- Интеграцию с комплексной обработкой событий (CEP)

Flink демонстрирует особую эффективность для сценариев, требующих сложных агрегаций в скользящих окнах или поддержки состояний в терабайтном диапазоне, что актуально для анализа социальных взаимодействий.

Apache Storm

Apache Storm относится к раннему поколению систем реального времени, обеспечивающему экстремально низкие задержки обработки (менее 5 мс). Его модель основана на направленных ациклических графах (DAG), где вершинами являются спауты (источники данных) и болты (обработчики). Storm оптимизирован для задач простой трансформации и маршрутизации потоков, но имеет ограничения в управлении состоянием и обработке временных меток. Для достижения семантики *"ровно один раз"* требуется использование надстройки Trident, что снижает производительность системы.

Spark Streaming

Spark Streaming реализует микропакетную модель обработки данных, преобразуя входящие потоки в последовательность RDD-датасетов (Resilient Distributed Datasets). Главным преимуществом данного подхода является унификация API для пакетной и потоковой обработки, позволяющая использовать единую кодовую базу для ETL-задач. Система обеспечивает:

- Интеграцию с экосистемой Spark (MLlib, GraphX)
- Гарантии *"ровно один раз"* через механизм чекпоинтов
- Поддержку обработки по времени событий

Ограничением Spark Streaming является повышенная задержка (от 500 мс до 2 с), обусловленная природой микропакетов.

RabbitMQ

RabbitMQ представляет собой традиционный брокер сообщений, реализующий протокол AMQP. В отличие от систем потоковой обработки, его фокус смещён в сторону гарантированной доставки и гибкой маршрутизации между очередями. Ключевые характеристики:

- Поддержка сложных схем маршрутизации (direct, topic, headers)
- Механизмы подтверждения доставки (acknowledgements)
- Интерфейсы управления очередями и сообщениями

Для задач аналитики в реальном времени RabbitMQ менее применим из-за отсутствия встроенных механизмов оконной агрегации и ограниченной пропускной способности по сравнению с Kafka.

Обоснование выбора

Для разрабатываемой системы анализа данных социальных сетей выбор пал на комбинацию Apache Kafka и Apache Flink. Kafka обеспечивает надёжную буферизацию входящих потоков из VK API с гарантией сохранности данных, в то время как Flink предоставляет необходимые инструменты для сложной агрегации и анализа с соблюдением требования к задержке < 60 секунд. Данная архитектура позволяет реализовать гибридную модель Каппа-архитектуры смешанной с микросервисной архитектурой, где единый поток данных обслуживает как реальную аналитику, так и пакетную обработку через реплей событий.

Таблица 1: Сравнительные характеристики систем обработки событий

Система	Тип. задержка	Гарантии доставки	Управление состоянием
Apache Kafka	< 10 мс	"ровно один раз"	Ограниченное
Apache Flink	50 мс	"ровно один раз"	Продвинутое
Apache Storm	< 5 мс	"минимум один раз"	Базовое
Spark Streaming	$0.5 - 2$ с	"ровно один раз"	Продвинутое
RabbitMQ	< 5 мс	"максимум один раз"	Отсутствует

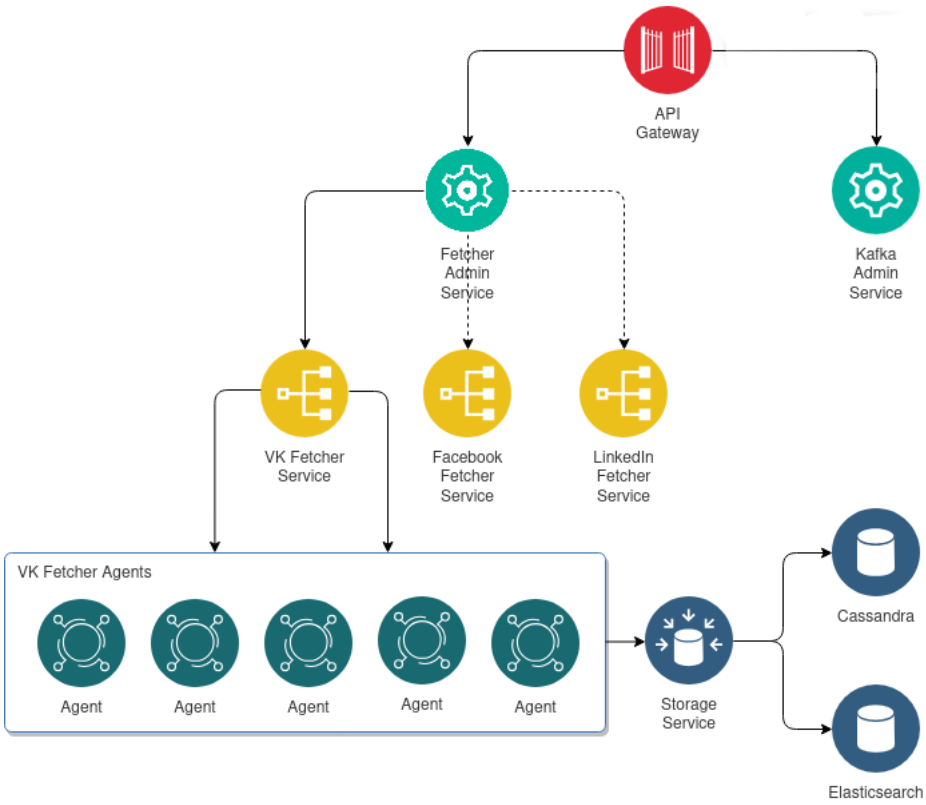

Рис. 1: Диаграма архитектуры

5 Разработанное решение

Как установлено в предыдущем разделе, проектируемая система требует одновременного обеспечения низкой задержки обработки, горизонтальной масштабируемости и устойчивости к сбоям. Реализация этих требований достигается через гибридную архитектуру, интегрирующую принципы Каппа-архитектуры с микросервисным подходом. Данный синтез формирует основу для обработки потоковых данных социальных сетей, обеспечивая баланс между производительностью, гибкостью и надёжностью.

Все микросервисы системы можно отнести к нескольким слоям: **Слой приёма данных**, **Слой обработки**, **Слой хранения и визуализации** и **Слой управления и администрации**. Каждый микросервис связан с другими через топики Kafka.

5.1 Слой управления и администрации

Слой управления и администрации является главным управляющим уровнем системы. На данным слое находятся сервисы отвечающие за непосредственное управление системой.

5.1.1 Взаимодействие сервисов

Каждый сервис взаимодействует с другоми через Kafka. Для этого используется формат JSON. Каждая команда имеет следующую структуру:

```
{
    "target_type":
    "content": {
    }
}
```

В поле `target_type` устанавливает получатель тип команды, используя который, система определяет какую из структур команды использовать и обрабатывает содержимое поля `content` соответствующим образом.

В рамках работы системы, для удобства обрыботки команд используется внутренняя структура данных, которая предварительно трансформируется в JSON и кодируется. Для создания команд используется внутренний класс-конструктор, а для трансформации - класс-обработчик. Эти классы изолированы и автоматически используются внутри специальзированной обёртка над Продюсером Kafka. Это позволяет снизить зависимости внутри системы и реализовывать взаимодействие системы без загромождения кода лишними вызовами обработчиков, что увеличивает читаемость кода и упрощает интеграцию в проект, а так же упрощает отладку. Для обратного декодирования из JSON во внутреннюю структуру используется библиотека `dacite`.

5.1.2 Kafka Admin Service

Kafka Admin Service — это сервис отвечающий за управление и администрирование Kafka. При инициализации он создаёт служебный топик и начинает обрабатывать приходящие в него команды. Основными ответственностями этого сервиса является создание и удаление топиков а так же их конфигурация, создание партиций. Через этот сервис проходит вся настройка работы Kafka.

Для взаимодействия с данных сервисом используется следующий формат `content`:

```
"content": {
    "command_type":
    "id":
    "topics_names": []
    "topics_parameters": {
        "name": []
    }
}
```

Где:

- `command_type` — тип полученной команды, обозначающий целевое действие (создание топика, удаление) и влияющее на то, какие поля запроса будут использоваться.
- `id` — Уникальный идентификатор запроса.
- `topics_names` — Список имён топиков.
- `topics_parameters` — Список параметров топиков. Для каждого топика из списка имён сдесь должны быть приведены соответсвующие параметры, а именно колличество партиций и уровень реплицирования.

5.1.3 Fetcher Admin Service

Fetcher Admin Service — это сервис отвечающие за администрацию сбора данных. Он не занимается сбором самостоятельно, а управляет сервисами по сборке данных из различных социальных сетей, например VK Fetcher Service. Основной обязанностью данного сервися является управление слоем приёма данных: добавление новых API токенов и удаление старых, управление списком обрабатываемых сообществ социальных сетей, добавление новых и удаление старых.

Для взаимодействия с данных сервисом используется следующий формат `content`:

```
"content": {
    "command_type":
    "id":
    "social_media_type":
    "groups": []
    "APIToken":
}
```

Где:

- `command_type` — тип полученной команды, обозначающий целевое действие (добавление API-токена, добавление сообществ в обработку) и влияющее на то, какие поля запроса будут использоваться.
- `id` — Уникальный идентификатор запроса.
- `social_media_type` — типо социальной сети, к которой отправляется запрос (VK, Twitter и тд)
- `groups` — Список групп / сообществ для добавления в обработку или удаления из неё.
- `APIToken` — Токен для добавления или удаления

5.2 Слой приёма данных

Слой приёма данных — это слой системы, но котором расположены сервисы, отвечающие за сбор данных и отправку их для дальнейшей обработки в топики Kafka. На этом уровне так же реализуется обработка всех команд, с учётом особенностей системы.

5.2.1 Особенности работы с VK

VK обладает открытым API, получить доступ к которому не составляет большого труда. Он предоставляет возможность для авторизации пользователя и собора открытой информации. Например такой информацией являются публикации в открытых группах, реакции на эти публикации и комментарии к ним. Так же с помощью Open API можно получить информацию о подписчиках сообществ, друзьях пользователей, фотографиях, видеозаписях и другой информацией, открытой пользователю. Иначе говоря VK API - это интерфейс, который позволяет получать информацию из базы данных vk.com с помощью HTTP-запросов к специальному серверу. Синтаксис запросов и тип возвращаемых ими данных строго определены на стороне самого сервиса. В ответ на запрос сервер возвращает ответ в формате JSON, что стоит учесть при выборе технологий, используемых для проектирования системы.

Для отправки API-запросов используется формат HTTP, в частности методы POST или GET. В API VK эти методы равнозначны. Запросы отправляются на адреса следующего вида:
`https://<адрес>/method/<API-метод>?<параметры>`

Где:

- `<адрес>` — один из адресов API ВКонтакте:
  - `api.vk.com`

– `api.vk.ru`

- `<API-метод>` — имя раздела и API-операции для вызова, например `users.get` или `likes.add`
- `<параметры>` — параметры, которые передаются методу в строке запроса, например `...?v=5.199&p1=v1`. Так же их называют query-параметрами.

VK API, как и API других социальных сетей, имеет ряд ограничений, которые стоит учитывать при проектировании системы:

1. Максимум 3 запроса / секунду с одного API-ключа.
2. Для `wall.get` возвращается ≤ 100 последних постов в сообществе.
3. Невозможность подписки на события в реальном времени.
4. Невозможность получения удалённых/скрытых постов.

5.2.2 VK Fetcher Service и VK Fetchers

VK Fetcher Service является центральным сервисом сбора данных социальной сети VK и управляется из Fetcher Admin Service. При инициализации он отправляет запрос в Kafka Admin Service на создание служебного топика для приёма команд от Слоя управления и администрации.

Для каждого из API-токенов VK Fetcher Service создаёт отдельный инстанс Фетчера, который в асинхронном порядке обрабатывает группы сообществ в обработке. Это нужно для того, что бы обойти ограничение на частоту запросов к API.

После инициализации сервиса, тот разделяется на 2 асинхронных потока, один из которых обрабатывает приходящие из Слоя управления команды, а второй создаёт очередь задач. Он начинает создавать задачи на обновление данных по конкретной группе из списка сообществ на обработку, после чего складывает их в очередь. Каждый из Фетчеров в свою очередь читает эти задачи и создаёт асинхронную задачу для разпараллеливания нагрузки, после чего уходит в ожидание на 0.5 секунды для соблюдения правил работы с API. Подобная архитектура позволяет уменьшить время простоев, одновременно регулируя и балансируя нагрузку на разные API-токены. Ещё одним приемуществом подобной архитектуры является возмжность динамически регулировать список обрабатываемых групп, а так же добавлять новые ключи, не останавливая работу сервису. Это крайне важно для регулирования больших всплесков активности и поддержания масштабируемости.

5.3 Слой обработки

Слой обработки выступает как главный рабочий слой. Он отвечает за предварительную обработку данных, их агрегацию и соверщение перобразований над ними (добавление данных, удаление ненужных и тд). Для этого используется пайплайн Kafka + Flink:

5.3.1 Интеграция с Kafka

Kafka выступает в роли шины событий, обеспечивая надежную передачу данных между микросервисами. Сообщения группируются по топикам в зависимости от типа

данных (например, посты, комментарии, внутренние команды). Для соблюдения упорядоченности данных ключи сообщений вычисляются на основе идентификатора группы, гарантируя последовательную обработку постов.

Внутри системы реализована специальная обёртка над Продюсером, которая инкапсулирует в себе предварительную обработку данных перед отправкой в топик (например преобразование в JSON). Так же перед отправкой данные кодируются в UTF-8 для обработки рускоязычного текста.

5.3.2 Интеграция с Flink

Apache Flink занимает центральное место в архитектуре системы, выступая в качестве высокопроизводительного движка потоковой обработки, который обеспечивает преобразование, агрегацию и анализ данных в режиме реального времени. Его интеграция в систему основана на принципах Каппа-архитектуры, что позволяет обрабатывать как текущие события, так и исторические данные через единый вычислительный конвейер, устраняя необходимость в раздельных системах для потоковой и пакетной обработки.

Flink выполняет критически важную функцию моста между источником данных (Apache Kafka) и системами хранения/аналитики (Elasticsearch и Cassandra). Основная задача Flink в системе — обеспечить обработку каждого поста из социальной сети VK с задержкой менее 60 секунд от момента публикации до сохранения результатов. Это достигается за счет непрерывной обработки событий, преобразований данных в реальном времени и управления состоянием вычислений при длительных операций.

Источником данных для Flink выступают топики Apache Kafka, куда VK Fetcher Service отправляет сырые данные из социальной сети. Flink потребляет эти данные через распределенного потребителя. Ключевой особенностью является использование механизма водяных знаков (watermarks), которые позволяют Flink определять полноту данных в потоке и корректно обрабатывать события, приходящие с задержкой из-за особенностей сети или неравномерной нагрузки в социальной сети.

Проходящие через Flink данные проходят следующие этапы преобразований:

1. **Нормализация и очистка**
   В первую очередь это приведение временных меток к единому формату ISO 8601, фильтрация некорректных или повреждённых записей, а так же стандартизация текстовых данных (приведение к одному регистру, нормальзация кодировки)

2. **богащение контекстом**
   На этом этапе добавляются дополнительные данные. Это добавление извлечение методанных, классификация тональности с помощью ML-моделей или любые дополнительные данные, которые могут помочь на следующем этапе.

3. **Оконная агрегация**
   На этом этапе выполняется расчет скользящих средних показателей (лайки, репосты) в 5-минутных окнах и обнаружение аномалий активности через статистические методы. Данный этап являтся главным для потоковой обработки.

Для управления задачами в кластере Flink используется job-manager-ui, доступ к котрому пробрасывается из кластера Flink

5.4 Слой хранения и визуализации

Данный слой не реализовывался в прототипе, однако стоит отдельно упомянуть. На данном слое располагаются сервисы ответственные за хранение и подготовку данных к

глубокому анализу и пакетной обработке.

5.4.1 Storage Service

Storage Service служит основным сервисом сохранения данных. Он читает полученные от Фетчеров данные, после чего сохраняет их в 2 хранилища: **Elasticsearch** (горячее хранилище) для быстрого поиска и дополнительной аналитики и **Cassandra** (холодное хранилище) для долгосрочного хранения и интеграции с глубокой, пакетной обработкой данных за большие промежутки времени.

5.4.2 Elasticsearch

Elasticsearch интегрирован в систему как ключевой компонент для оперативной аналитики, обеспечивающий полнотекстовый поиск, агрегацию данных и визуализацию в режиме реального времени. В архитектуре системы он выполняет роль горячего хранилища и основного движка аналитики. Опционально Elasticsearch так же можно использовать для визуализации через интеграцию с Kibana. Индексы настроены с анализаторами для русскоязычного контента. Агрегации по временным гистограммам позволяют визуализировать активность пользователей. Для сохранения данных в Elasticsearch используется официальный коннектор Flink `org.apache.flink.connector.elasticsearch7`

Данные поступают из Apache Flink через REST API с использованием bulk-запросов для оптимизации производительности. Для поддержания уникальности используется состовной ключ `group_id_post_id`, который гарантирует уникальность записей.

5.4.3 Cassandra

CApache Cassandra в системе выполняет роль высокомасштабируемоего распределенного хранилища для долгосрочного хранения больших объемов данных. В архитектуре системы она выполняет функции основного хранилища для сырых данных и агрегированных показателей, платформы для исторического анализа данных за длительные периоды и системы резервного копирования. Схема таблиц может быть оптимизирована для высокой скорости записи, а горизонтальная масштабируемость достигнута без изменения архитектуры добавлением новых нод.

Ключевое пространство social_media может быть организовано с учетом специфики социальных данных. Это достигается за счёт стратегии репликации: NetworkTopologyStrategy обеспечивает копирование данных между дата-центрами (например, 3 копии в основном DC, 2 - в резервном), а так же проектированием таблиц с опорой на особенности данных. Основные таблицы:

- **raw_posts** — Хранит исходные данные постов с партиционированием по ID группы
- **daily_aggregates** — Содержит предварительно агрегированные дневные метрики (количество постов, средние лайки/комментарии)
- **backups** — Архивные копии данных из Elasticsearch

Для повышения безопасности и отказоустойчивости, данные раз в несколько дней копируются из Elasticsearch и распределяются по дата-центрам, каждая копия содержит метку времени и идентификатор источника.

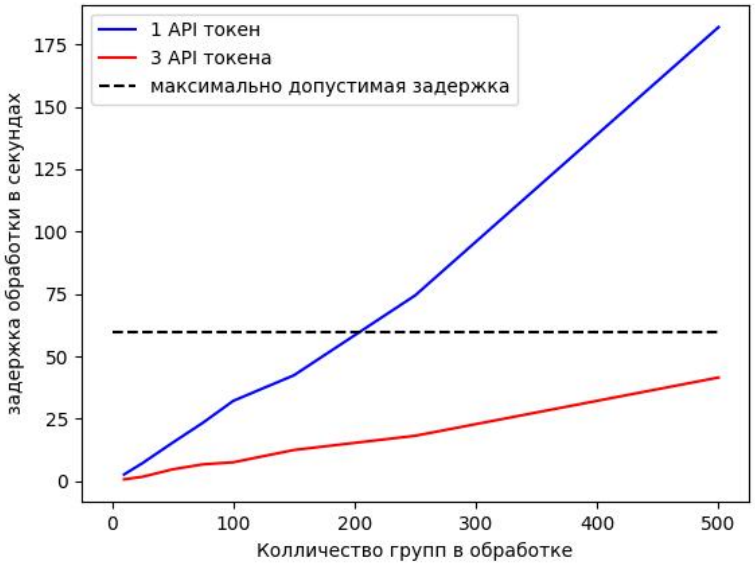

Рис. 2: График задержки обработки

5.5 K8S

Для упрощения запуска системы, её развёртывания и масштабирования, для каждого сервиса был написан Dockerfile и yml-файл, что позволяет запустить систему в K8S. Это значительно упрощает управление системой, открывает доступ к реализации автоматического масштабирования любого сервиса и оркестризации. Так же это упрощает интеграцию с Flink и Kafka, за счёт использования уже готовых Docker-образов.

5.6 Соответствие требованиям

Соответствие требованиям подтверждено замерами задержки в обработке сообществ ВК. Замерялось время обработки всех сообществ из списка обработки. Для удобства анализа данные изображены на графике.

Как можно заметить главным узким местом системы, как и предполагалось является отправка и обработка запросов VK. Однако при масштабировании колличества API-токенов, система автоматически использует новые ресурсы и задержка значительно снижается. Таким образом для обработки 500 групп достаточно 3-х API-токенов, а для обработки 10000 сообществ с задержкой менее минуты, достаточно 50 токенов.

Так же были сняты замеры времени обработки одного сообщества. Для прохождения всего пайплайна сообществом, системе необхадимо 0.4 секунды.

5.7 Исследование системы на предмет улучшений и ускорения

Для разработанной системы был проведён анализ узких мест и предложены варианты для их улучшения и оптимизации:

1. Одним из самых узких мест в системе оказалась ситема фетчинга данных социальных сетей. Для решения этой предлагаются следующие улучшения: автоматическое удаление недоступных и закрытых сообществ, оптимизация обновления данных группы, основываясь на частоте публикаций в сообществе. Нарпимер сообщество которое публикует новости раз в сутки менее целесообразно опрашивать каждую минуту, в отличии от сообществ, публикующих данные каждые 10 минут. Динамически настраивая подобные параметры, можно добиться ускорения

в несколько раз. Однако всё ещё нужно оставить возможность отключения этой функции для конкретных сообществ, для сообществ, требующих максимально высокой точности и низкой задержки.

2. Ускорение Flink-Обработки возможно достичь с помощью предварительной агрегации данных и оптимизации состояний, а так же векторизации обработки. таким образом можно будет достичь сокращение времени обработки окон на $25 - 35\%$

3. Оптимизация Доступа к Данным может стать необхадима при увеличении колличества обрабатываемых данных. Для Elasticsearch этих оптимизаций можно достичь за счёт гибридного кэша в Radis (самые активыне сообщества и пользователи) и прекомпиляции шаблонов запросов.

4. В случае необхадимости оптимизации Kafka, например при росте задержек при малом размере сообщений, можно воспользоваться сжатием данных, перед отправкой в Kafka, например при помощи настройки Kafka (compression_type="snappy") или использованием zstd

6 Заключение

Настоящее исследование посвящено проектированию, разработке и реализации прототипа системы потоковой обработки данных социальных сетей, ориентированной на анализ пользовательского контента в режиме реального времени. В ходе работы достигнуты следующие ключевые результаты:

1. Предложена **гибридная архитектура**, интегрирующая принципы **Kappa-архитектуры** (единый поток событий, репроцессинг данных) с **микросервисным подходом**. Это позволило ликвидировать дублирование логики обработки и обеспечить модульность и независимое масштабирование компонентов.

2. Разработан **механизм сбора данных** VK на базе асинхронного фреймворка **vkbottle**, обеспечивающий:

   - Обход ограничений rate limit (3 запроса/секунду) через динамическое распределение задач по API-токенам
   - Минимизацию задержек за счёт оптимизированной очереди задач
   - Гибкое управление источниками через Fetcher Admin Service

3. Реализован **прототип конвейера обработки** на базе **Apache Kafka** (шина событий с "ровно один раз"семантикой) удовлетворяющий требованию задержки $<$ 60 секунд. Исследована возможность подключения к конвееру Apache Flink для потоковой обработки и анализа данных

   Исследована и разработана архитектура, учитывающая **двухуровневую систему хранения**:

   - **Elasticsearch** для оперативного поиска и визуализации
   - **Cassandra** для долгосрочного хранения и интеграции с пакетной аналитикой

6.1 Практическая значимость

Практическая значимость работы заключается в создании универсальной платформы для маркетингового анализа (тренды, репутация брендов), социологических исследований (динамика настроений) и кризисного мониторинга (детекция аномалий)

6.2 Перспективные направления развития

: Работа имеет перспективные направления для развития в области Обработки данных. В качестве направлений развития выделены следующие варианты:

- Добавление поддержка Telegram/Twitter через модули Fetcher Admin Service
- Полноценная интеграция Apache Flink для потоковой обработки детализированных
- Реализация хранилищ данных на основе исследованных вариантов.
- Исследование использования шардирования в Cassandra
- Внедрение ML-пайплайнов для классификации контента
- Оптимизация Flink-операторов для снижения задержки до 10-20 секунд

- Исследование методов сжатия данных в Kafka/Cassandra

Таким образом, разработанное решение подтверждает эффективность гибридного подхода для задач потоковой обработки Big Data, сочетая оперативность Kappa-архитектуры с гибкостью микросервисов, и открывает пути для дальнейшей оптимизации в условиях экспоненциального роста социальных данных.

Список литературы

[1] *Apache Software Foundation.* — Apache Kafka Documentation. — Apache Software Foundation, https://kafka.apache.org/documentation/, Latest version edition, 2023. — October.

[2] *Apache Software Foundation.* — Apache Flink Documentation. — Apache Flink Project, https://nightlies.apache.org/flink/flink-docs-stable/, Version 2.0 edition, 2024. — December.

[3] *Apache Software Foundation.* — Apache Spark Documentation. — Apache Spark Project, https://spark.apache.org/docs/latest/, Version 3.5 edition, 2024. — November.

[4] *Apache Software Foundation.* — Apache Storm Documentation. — Apache Storm Project, https://storm.apache.org/documentation/, Latest edition, 2023. — August.

[5] *Confluent Inc.* — Kafka Streams Documentation. — Confluent Platform Documentation, https://docs.confluent.io/platform/current/streams/index.html, Confluent Platform 7.3 edition, 2023. — July.

[6] *Cloud Native Computing Foundation.* — Kubernetes Documentation. — Kubernetes Project, https://kubernetes.io/docs/, v1.28 edition, 2024. — March.

[7] *Mikhail Borisov.* — vkbottle Documentation. — GitHub Repository, https://vkbottle.readthedocs.io/ru/latest/, v3.5.0 edition, 2024. — July.

[8] *Elastic B.V.* — Elasticsearch Reference Documentation. — Elastic, https://www.elastic.co/docs/solutions/search, v8.11 edition, 2023. — December.

[9] *Apache Software Foundation.* — Apache Cassandra Documentation. — Apache Cassandra Project, https://cassandra.apache.org/doc/latest/, 4.1 edition, 2024. — January.

[10] *Zaharia, Martin.* Designing Data-Intensive Applications / Martin Zaharia, Benjamin Wenkel. — O'Reilly Media, 2021.

[11] Learning Spark: Lightning-Fast Big Data Analysis / Holden Karau, Andy Konwinski, Patrick Wendell, Matei Zaharia. — O'Reilly Media, 2022.

[12] *Gourley, David.* High Performance Browser Networking / David Gourley, Brian Totty. — O'Reilly Media, 2013.

[13] *Narkhede, Neha.* Kafka: The Definitive Guide / Neha Narkhede, Brandon Pitt, Gwen Shapira. — O'Reilly Media, 2017.

[14] *Friedman, Tyler.* Streaming Systems: The What, Where, When, and How of Large-Scale Data Processing / Tyler Friedman, Tyler Akidau, Slava Chernyak. — O'Reilly Media, 2018.

[15] *Chappell, David.* Enterprise Integration Patterns / David Chappell, Mark Richardson. — Addison-Wesley, 2003.

[16] Accelerate: Building and Scaling High Performing Technology Organizations / Gene Kim, Jez Humble, Patrick Debois, John Willis. — IT Revolution Press, 2018.

[17] *Leskovec, Jure.* Mining of Massive Datasets / Jure Leskovec, Anand Rajaraman, Jeffrey D. Ullman. — Cambridge University Press, 2020.

[18] Social Media Analytics: Mining, Modeling, and Visualization of Big Data / Muhammad Umar Farooq, Muhammad Rizwan Asghar, Waqas Ahmad, Asad Waqar Khan. — Institute of Electrical and Electronics Engineers (IEEE), 2021.

[19] *Kreps, Jay.* I love Logs: Ideas from the Founders of Kafka, Samza, and Crest / Jay Kreps. — O'Reilly Media, 2014.

[20] *Red Hat.* What is Apache Kafka? / Red Hat // *Red Hat Product Documentation.* — 2022. — October.

[21] *Decodable Inc.* Introduction to Apache Flink and Stream Processing / Decodable Inc. // *Decodable Blog.* — 2022. — December.

[22] *Spiceworks.* What Is Kafka? Architecture and Uses Explained / Spiceworks // *Spiceworks Technology Network.* — 2022. — July.

[23] *Upsolver.* Apache Kafka Use Cases: When To Use It? When Not To? / Upsolver // *Upsolver Engineering Blog.* — 2023. — January.

[24] *Apache Software Foundation.* Потоковая обработка данных с Apache Flink / Apache Software Foundation // *PDF-книга / Руководство.* — 2021.

[25] *Anonymous.* Работа с BigData в облаках. Обработка и хранение / Anonymous. — Эко-Пресс, 2021.

[26] *Anonymous.* Apache Kafka: потоковая обработка и анализ данных / Anonymous. — Диалектика, 2022.

[27] *Kubernetes Contributors.* — Kubernetes Documentation: Official Guide. — Cloud Native Computing Foundation (CNCF), https://kubernetes.io/docs/home/, 2025 Edition edition, 2025. — June.

[28] *Джиджи, Сайфан.* Осваиваем Kubernetes: Оркестрация контейнерных архитектур / Сайфан Джиджи. — Manning Publications, 2019.

[29] *Самсонов, Алексей.* Kubernetes для разработчиков / Алексей Самсонов. — ДМК Пресс, 2021.

[30] *Хасс, Билджин Ибрам и Роланд.* Паттерны Kubernetes / Билджин Ибрам и Роланд Хасс. — O'Reilly Media, 2020.

[31] *Proust, Daniel.* The Kubernetes Book / Daniel Proust. — Apress, 2017.