

УДК 004.41:004.75

АВТОМАТИЗИРОВАННЫЙ СБОР ДАННЫХ ИЗ ОТКРЫТЫХ ИСТОЧНИКОВ МАССОВОЙ ИНФОРМАЦИИ: ПРОЕКТИРОВАНИЕ РАСПРЕДЕЛЁННОЙ АРХИТЕКТУРЫ С ДЕКЛАРАТИВНЫМ ОПИСАНИЕМ ИСТОЧНИКОВ И ПРОВЕРЯЕМОЙ ФИКСАЦИЕЙ ЦЕЛОСТНОСТИ КОРПУСА

Владимирова Т.Р., педагог дополнительного образования кафедры русского языка № 1, Институт русского языка, Российский университет дружбы народов (117198, Москва, ул. Миклухо-Маклая, д. 6), e-mail: tvladimirovar@gmail.com

Лишина П.В., лаборант кафедры русского языка № 3, Институт русского языка, Российский университет дружбы народов (117198, Москва, ул. Миклухо-Маклая, д. 6), e-mail: lishina-pv@rudn.ru

Лесько С.А., д.т.н., профессор кафедры КБ-2 «Информационно-аналитические системы кибербезопасности», институт кибербезопасности и цифровых технологий, МИРЭА - Российский технологический университет (119454, г. Москва, проспект Вернадского, дом 78), e-mail: lesko@mirea.ru

Регулярный сбор публикаций из открытых источников массовой информации осложнён неоднородностью их программных интерфейсов, изменчивостью разметки страниц и необходимостью подтверждать происхождение и неизменность накопленного корпуса. Цель работы – спроектировать и экспериментально проверить архитектуру приложения, в которой подключение источника не затрагивает программное ядро, полнота периодического опроса лент контролируется при переменной интенсивности публикаций, а состояние собранного корпуса допускает последующую проверку. Приложение построено из крупных функциональных модулей, взаимодействие которых организовано через брокер сообщений с устойчивой доставкой «не менее одного раза» (at-least-once). Для хранения оперативного состояния используется распределённый кэш. Описание источника учитывается по версиям и применяется при повторном разборе сохранённых ответов. Глубина периодического опроса ленты определяется перекрытием окон публикаций, а проверка состояния корпуса выполняется по дереву Меркла и подписанным контрольным точкам. Приложение сопоставлено со Scrapy при одинаковой степени параллелизма: для конфигураций с одним–восемью исполнителями отношение средних пропускных способностей находилось в пределах 1,70–2,00. После управляемого изменения структуры сохранённых HTML двух новостных ресурсов повторный разбор восстановил значения ранее извлекавшихся полей. Новых HTTP-запросов для этого не потребовалось. В каждом из трёх отказных эпизодов все документы были сохранены, устойчивые дубликаты отсутствовали. Для дерева из миллиона записей путь включения занимал в среднем 640 байт; его проверка выполнялась за 24,50 мкс. Эти результаты характеризуют испытанную реализацию в условиях одноузлового стенда.

Ключевые слова: автоматизированный сбор данных, открытые источники массовой информации, распределённая архитектура, брокер сообщений, распределённый кэш, декларативный шаблон извлечения, XPath, перекрывающийся опрос, дерево Меркла, контроль целостности, отказоустойчивость.

AUTOMATED DATA COLLECTION FROM OPEN MASS MEDIA SOURCES: DESIGN OF A DISTRIBUTED ARCHITECTURE WITH DECLARATIVE SOURCE DESCRIPTIONS AND VERIFIABLE CORPUS INTEGRITY ATTESTATION

Vladimirova T.R., Teacher of Supplementary Education, Department of Russian Language No. 1, Institute of Russian Language, RUDN University (6 Miklukho-Maklaya St., Moscow, 117198, Russia), e-mail: tvladimirovar@gmail.com

Lishina P.V., Laboratory Assistant, Department of Russian Language No. 3, Institute of Russian Language, RUDN University (6 Miklukho-Maklaya St., Moscow, 117198, Russia), e-mail: lishina-pv@rudn.ru

Lesko S.A., Doctor of Technical Sciences, Professor at the Department KB-2 “Information and Analytical Cybersecurity Systems”, Institute of Cybersecurity and Digital Technologies, MIREA - Russian Technological University (78 Vernadsky Avenue, Moscow, 119454, Russia), e-mail: lesko@mirea.ru

Regular collection of publications from open mass media sources is complicated by the heterogeneity of source interfaces, the volatility of page markup and the need to attest the provenance and immutability of the accumulated corpus. The aim of the study is to design and experimentally verify an application architecture in

which adding a source does not affect the program core, the completeness of periodic feed polling is monitored under a variable publication rate, and the state of the collected corpus remains verifiable. A combined architecture is proposed, in which coarse-grained functional modules are complemented by a coordination layer formed by a message broker with durable, at-least-once delivery and a distributed cache of short-lived state. The application combines a versioned declarative source template with reparsing of stored responses, adaptive feed polling based on overlapping publication windows, and verifiable corpus attestation using a Merkle tree and signed checkpoints. These mechanisms form a reproducible sequence linking source responses, extraction rules and normalised records. The experimental implementation was compared with Scrapy at matching process counts. With one to eight workers, the ratio of mean batch throughputs ranged from 1.70 to 2.00. A separate experiment used a shared request-rate limiter to characterise operation under a constrained input flow. Stored HTML pages from two news resources were subjected to a controlled local markup transformation and parsed using revised templates. No additional HTTP requests were sent during reparsing; the non-empty field values obtained with the original template were recovered without differences. Message redelivery occurred in the worker, RabbitMQ and Redis failure tests, but each of the three episodes ended with all unique results stored. With one million records in the tree, the mean inclusion-path size was 640 bytes and verification took 24.50 microseconds. The reported timings were obtained on one node for the stated software versions, using the measurement intervals specified for each experiment.

Keywords: automated data collection, open mass media sources, distributed architecture, message broker, distributed cache, declarative extraction template, XPath, overlapping polling, Merkle tree, integrity control, fault tolerance.

Введение

Для мониторинга событий, изучения общественного мнения и выявления информационных кампаний собирают публикации новостных сайтов, блогов, новостных лент и социальных платформ. Тексты и мультимедийные материалы этих ресурсов входят в тематические корпуса и используются в системах поддержки принятия решений. При работе с ними необходимо проверить сам сбор: какие документы получены, насколько полно охвачен источник, можно ли воспроизвести процедуру. Оценка обоснованности выводов требует не только сведений о методах анализа, но и описания того, как были получены исходные материалы.

Единогo способа загрузки для всех перечисленных источников нет. Статический HTML разбирается после HTTP-запроса; RSS и JSON имеют собственное структурированное представление. Если же страница формируется сценариями JavaScript, это приходится учитывать при получении её содержимого. Схему обхода выбирают с учётом глубины архива, постраничной выдачи, частоты обновлений и доступных прикладных программных интерфейсов. Самой загрузкой работа не ограничивается: сборщик должен сохранять медиафайлы, обрабатывать сетевые сбои, отслеживать изменения разметки и выдерживать заданную частоту запросов.

Однопроцессную программу удобно использовать для разовой выборки: её сравнительно легко разработать и проверить. При непрерывном опросе многих ресурсов приходится дополнительно управлять очередью, исключать повторную обработку одних и тех же адресов и согласовывать использование сети и диска. Кроме того, отказ этого процесса прерывает весь сбор. В рассматриваемой архитектуре обработчик не совмещает все перечисленные функции. Управление заданиями, загрузка страниц, извлечение полей, запись результатов и обработка ошибок распределены между компонентами приложения.

Правила извлечения приходится пересматривать вслед за разметкой сайта — например, после смены классов HTML-элементов или перестановки блоков. Когда правила включены в код сборщика, такие изменения и подключение нового ресурса требуют правки программы. В разработанном приложении они хранятся отдельно, в конфигурационном шаблоне. Вместе с правилами поиска полей шаблон задаёт параметры доступа, условия применимости к странице и области, которые следует исключить из обработки. Каждая редакция получает свою версию; сведения о разметке конкретного ресурса остаются за пределами программного ядра.

Размер окна последних публикаций влияет и на объём чтения, и на риск пропусков. Если между двумя опросами появилось больше записей, чем помещается в окно, часть из них не попадёт в очередную выборку. Можно каждый раз перечитывать всю ленту, но тогда возрастает нагрузка на источник. Здесь используется другой порядок: сборщик сравнивает новое окно с предыдущим и ищет уже известные публикации. Если их недостаточно, он продолжает чтение вглубь ленты до достижения требуемого перекрытия.

Документ, помещённый в реляционную базу, по-прежнему можно изменить или удалить штатными средствами. Само хранение в базе поэтому не подтверждает соответствия документов ранее собранному корпусу. Для такой проверки в приложении ведётся журнал целостности. В используемой схеме канонические записи объединены в упорядоченное дерево Меркла; его состояние фиксируют подписанные контрольные точки, копии которых находятся вне основного контура. По этим копиям проверяется соответствие корпуса

ранее зафиксированному состоянию. Развёртывание блокчейн-инфраструктуры для этой задачи потребовало бы избыточных вычислительных и инфраструктурных затрат.

Цель работы — спроектировать и экспериментально проверить архитектуру приложения для регулярного сбора публикаций из открытых источников массовой информации. При проектировании поставлены три требования: новые источники подключаются без изменения программного ядра; полнота периодического опроса контролируется при изменении интенсивности публикаций; состояние собранного корпуса можно проверить после его фиксации. Для достижения цели решены четыре задачи: разработана декларативная модель описания источника; определены состав компонентов и порядок их взаимодействия, обеспечивающий однократную фиксацию результата при повторной доставке задания; реализованы алгоритм перекрывающегося опроса ленты и правило назначения заданий разнородным исполнителям; построена схема проверяемой фиксации состояния корпуса.

Научно-прикладной вклад работы состоит в согласовании механизмов воспроизводимого сбора и проверяемого хранения корпуса. Во-первых, декларативный шаблон источника связан с версией правил и сохранённым исходным ответом, что позволяет повторно выполнять разбор после исправления правил, не обращаясь к источнику. Во-вторых, для периодического опроса ленты задан критерий достаточности пересечения соседних окон, по которому глубина обхода изменяется адаптивно. В-третьих, определён порядок фиксации результатов и контрольных точек, связывающий повторную доставку заданий, идемпотентную запись и проверку состояния корпуса. Для выбранных вариантов реализации получены экспериментальные оценки производительности, поведения при отказах и затрат контроля целостности.

1. Анализ существующих решений и выбор архитектурного подхода

Современное состояние средств автоматизированного извлечения данных из веб-ресурсов систематизировано в обзорах [1–3], где методы сопоставлены по производительности, устойчивости к изменению разметки, стоимости сопровождения и воспроизводимости результата. Прикладные исследования [4, 5] показывают, что качество извлечения определяется явным описанием структуры источника, а адаптивное восстановление правил требует обучающих данных и не отменяет необходимости хранить шаблон в интерпретируемом виде.

В отечественных работах рассмотрены автоматическое построение правил извлечения для новостных сайтов [6], высокопроизводительное распределённое средство сбора [7], система управления заданиями автоматизированного сбора [8], хранение исходных и нормализованных данных [9], архитектура поисково-аналитической системы [10] и классификация HTTP-запросов при автоматическом обходе современных веб-приложений [11]. Совокупность этих результатов обосновывает разделение обнаружения адресов, загрузки, извлечения, хранения и контроля состояния между самостоятельными компонентами.

В работе [12] рассмотрено декларативное задание правил извлечения конечным пользователем. Такой подход позволяет сохранять состав полей и селекторы в явном виде, доступном для проверки и изменения. В рассматриваемом приложении для этого используются шаблоны YAML. При определении границ компонентов учитывались результаты архитектурных обзоров [13, 14]: независимое масштабирование отдельных частей приложения полезно, но чрезмерное дробление увеличивает число взаимодействий и затрудняет согласование данных.

Выбор вычислительной структуры и наблюдаемость распределённых приложений исследованы в [15, 16]. Для сбора новостного материала в [17] предложена архитектура с распределённым реестром; круг решаемых в ней задач шире проверки неизменности локально накопленного корпуса. Работы [18–20] рассматривают фиксацию происхождения данных и защищённое журналирование. При интерпретации локальных измерений производительности использованы рекомендации [21].

Общая очередь и несколько экземпляров сборщика применяются в scrapy-redis [22] и Scrapy Cluster [23]. Для сохранения исходных веб-ответов и их последующего использования предназначен формат WARC [24]. Построение проверочных путей и контрольных значений в настоящей работе опирается на RFC 9162 [25]. Назначение заданий разнородным исполнителям выполняется по эвристике минимального времени завершения МСТ [26], а организация периодического обхода рассматривается с учётом политик обновления веб-копий [27]. Эти подходы соотносятся с учётом версий шаблонов и последовательностью формирования корпуса.

На рисунке 1 представлены три архитектурных подхода по [28]: микросервисный, сервисный (Service-Based) и основанный на распределённом пространстве данных. В микросервисном варианте компоненты имеют собственные хранилища и независимые жизненные циклы. Сервисный подход предполагает более крупные функциональные блоки, связанные общей интеграционной средой. В варианте с распределённым пространством данных очередь заданий, оперативное состояние и распределение нагрузки размещаются в виртуализированном промежуточном слое.

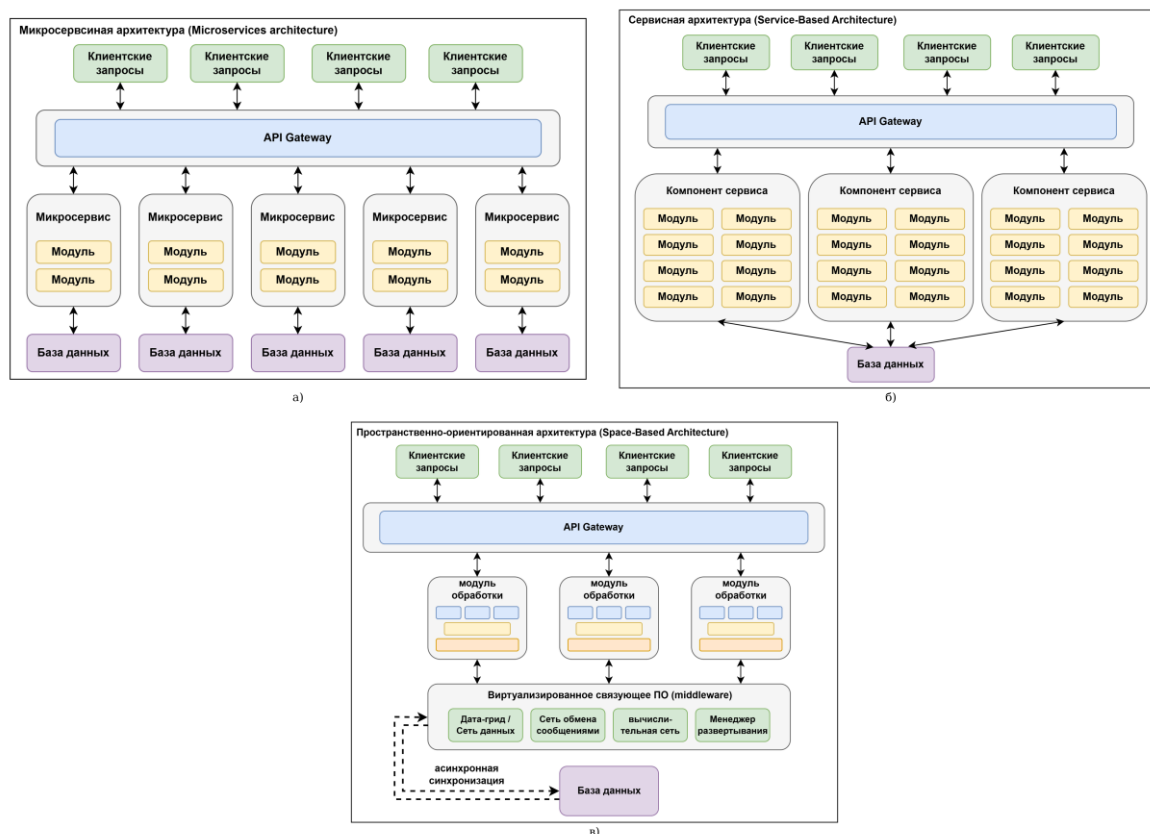


Рисунок 1 – Архитектурные подходы: а) микросервисный; б) сервисный (Service-Based); в) основанный на распределённом пространстве данных. Схемы по [28]

Для рассматриваемой задачи предложена комбинированная архитектура. В ней сохранены крупные функциональные компоненты сервисного подхода, а координация вынесена в общий слой: брокер сообщений доставляет задания, распределённый кэш хранит оперативное состояние, обработку ведёт группа исполнительных узлов. Такое сочетание сохраняет отчётливые границы компонентов и позволяет наращивать число исполнителей независимо от остальных частей приложения.

Анализ существующих решений позволяет выделить три требования, которые определяют дальнейшее изложение: подключение источника не должно затрагивать программное ядро; периодический опрос ленты должен сохранять полноту при переменной интенсивности публикаций; собранный корпус должен допускать проверку происхождения и неизменности документов. Способы выполнения этих требований рассмотрены в разделах 2–4.

2. Декларативная модель источника и правила извлечения

Модуль извлечения получает загруженный документ и шаблон источника, после чего возвращает набор полей; поиск адресов ему не поручается. Адреса заранее собирает модуль обнаружения, просматривающий разделы сайта, карты сайта, ленты RSS и прикладные программные интерфейсы. Благодаря такому разделению правила извлечения пересматриваются без изменения обхода ссылок, а единый механизм обнаружения применяется к разным источникам.

Выбор правил зависит не только от формата ответа, но и от типа источника; четыре характерные группы приведены в таблице 1. Для новостного портала, как правило, удаётся задать устойчивый набор полей, хотя отдельные разделы и специальные проекты требуют собственных шаблонов. У блогов структура страниц различается сильнее, а обновления менее регулярны. Записи лент RSS и Atom имеют готовую структуру, но часто содержат лишь краткое описание со ссылкой на полный текст. Социальные платформы рассматриваются отдельно, поскольку помимо веб-страниц и программных интерфейсов здесь приходится учитывать ограничения доступа.

Таблица 1 – Программные интерфейсы, способы получения данных и ограничения источников

Тип источника	Программный интерфейс	Способ получения данных	Обновление	Ограничения
Новостный портал	HTML, RSS, иногда прикладной интерфейс	HTTP-запрос и разбор по XPath или CSS, при необходимости браузерная обработка	регулярное, несколько шаблонов	смена разметки, постраничная выдача, ограничение частоты запросов
Блог	HTML, RSS	HTTP-запрос, извлечение по CSS, XPath и регулярным выражениям	нерегулярное	различия в структуре страниц и метаданных
Лента новостей	RSS, Atom, XML, JSON	последовательное чтение элементов ленты	регулярное	малая глубина выдачи, сокращённый текст публикации
Социальная платформа	прикладной интерфейс, JSON, динамический HTML	прикладной интерфейс, разрешённые сетевые запросы или браузерная обработка	интенсивное и неравномерное	авторизация, квоты, смена интерфейса

Описание каждого источника вынесено в шаблон формата YAML; пример приведён в листинге 1. Исполняемый код в шаблоне отсутствует, поэтому его проверка, изменение и учёт версий выполняются независимо от приложения. Шаблон содержит метаданные источника и параметры доступа, признаки применимости, правила удаления служебных областей, перечень полей с указанием типов данных и правила дополнительного поиска, а также расписание сбора и параметры сохранения медиафайлов.

Перед применением шаблон проходит синтаксический разбор YAML, компиляцию регулярных выражений и XPath, затем проверку на сохранённом примере страницы. Регулярные выражения в листинге записаны в одинарных кавычках, сохраняющих обратную косую черту [31].

Перед извлечением полей обработчик проверяет условия группы conditionElement. Наличие перечисленных в ней обязательных элементов означает, что страница может быть разобрана по выбранному шаблону; при их отсутствии документ получает состояние «не соответствует шаблону» и не обрабатывается как обычная публикация. От сетевой ошибки это состояние отличается тем, что ответ источника уже получен и после исправления шаблона может быть разобран повторно.

Перед выделением основного текста удаляются области, перечисленные в группе ignoreElement: сценарии, рекламные вставки, навигация, блоки рекомендаций и другие служебные фрагменты. Состав результата описывает группа parseElement, где для заголовка, вводного текста, автора, даты, основного текста и медиафайлов заданы селекторы и типы получаемых значений. Дополнительный поиск по документу вынесен в группу specialElement; к нему относится, например, выделение адресов электронной почты, имён персон и других сущностей, распознаваемых по образцу.

Способ поиска выбирается в зависимости от того, как поле представлено в документе. Выражение XPath описывает положение элемента в дереве документа и условия по его атрибутам; для элементов с устойчивыми классами и идентификаторами короче и нагляднее CSS-селекторы. Ссылки и медиафайлы извлекаются селекторами по атрибутам, а регулярные выражения применяются к уже выделенному тексту и не заменяют разбор HTML целиком. Сочетание этих средств позволяет описать требуемые правила, сохраняя шаблон пригодным для чтения и проверки человеком.

Листинг 1 – Конфигурационный шаблон извлечения для новостного сайта (пример)

```
template:
  name: Новостной источник (пример)
  url: https://example.org/
  sourceType: News portal
  schedule: null
  robotCountry: null
  typeOfRobot: Requests
  proxy: null
  userAgent: 'ResearchCrawler/1.0 (+contact@example.org)'
  downloadMedia: false
  robotsTxt: true
  conditionElement:
    - name: article element
      selector: XPath
```

```

    value: //div[@id='root']/main[@id='main-content']
ignoreElement:
- name: script
  selector: XPath
  value: //script
- name: similar articles
  selector: XPath
  value: //aside[@class='article__similar-articles']
parseElement:
- name: title
  selector: XPath
  value: //h1[contains(@class, 'article-headline__text')]
  type: text
- name: intro
  selector: XPath
  value: >-
    //div[@class='article__body-content']/div[contains(@class,
    'article__intro')]
  type: text
- name: author
  selector: XPath
  value: //meta[@name='author']/@content
  type: text
- name: date
  selector: XPath
  value: //time/@datetime
  type: datetime
- name: body
  selector: XPath
  value: //div[contains(@class, 'body-text-card__text')]//p
  type: text
- name: media
  selector: XPath
  value: //article//img/@src
  type: media
specialElement:
- name: email
  selector: regex
  value: '\b[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}\b'
  type: email
- name: person
  selector: regex
  value: '\b(?:Mr\.?|Mrs\.?|Ms\.?|Dr\.?)\s+([A-Za-z]+\s+[A-Za-z])\b'
  type: text

```

Соответствие полей шаблона областям новостной страницы показано на рисунке 2: отмечены автор, дата, вводный абзац (лид) и основной текст, а приведённое выражение XPath указывает на абзацы внутри блока содержимого.

The image shows a news article layout with several elements highlighted and labeled:

- автор** (author): Points to the byline "By Jessica Bradley".
- дата** (date): Points to the date "8th January 2024".
- лид** (lead): Points to the first paragraph of the article: "Seaweed has long been a staple of many cultures, but the Western diet seems to have forgotten it. Is it a healthy ingredient we should be embracing?"
- текст** (text): Points to the main body of the article, which includes several paragraphs about seaweed.

A box labeled **Пример XPath** (Example XPath) contains the following expression:

```
//div[contains(@class, 'body-text-card__text')]//p
```

Рисунок 2 – Соответствие областей новостной страницы полям шаблона

Статические страницы приложение получает прямым HTTP-запросом. Если известно, что ресурс формирует содержимое средствами JavaScript, отсутствие обязательных элементов служит основанием передать задание браузерному обработчику. Вместо запуска браузера нередко удаётся воспользоваться разрешённым интерфейсом JSON либо тем запросом Fetch или XHR, который страница выполняет при обычном просмотре. Браузерная обработка применяется только там, где она необходима, поскольку увеличивает время выполнения и расход оперативной памяти.

В шаблоне задаются и параметры выполнения: расписание, необходимость сохранять медиафайлы, регион исполнения запроса, настройки идентификатора клиентской программы и порядок учёта файла robots.txt. При создании задания планировщик принимает во внимание ограничения источника, в том числе допустимую частоту обращений.

Сбор выполняется с учётом правил доступа к ресурсу и robots.txt [29], с постоянным идентификатором клиентской программы и ограниченной частотой запросов. Авторизация и технические ограничения не обходятся. Извлечение персональных данных включается только при обоснованной исследовательской необходимости; токены, cookies и другие секреты исключаются из распространяемых копий ответов.

Полноту извлечения оценивают по карте покрытия источника, в которой для каждого поля – заголовка, даты, автора, основного текста и медиафайлов – указана доля документов, где это поле найдено. Снижение доли служит основанием проверить разметку и правила разбора. Для каждой публикации сохранены исходный ответ и идентификатор применённой версии шаблона, поэтому после исправления правил ответ разбирают повторно, не обращаясь к сайту.

3. Архитектура приложения и алгоритмы обработки

Состав компонентов целевой архитектуры приложения и связи между ними приведены на рисунке 3. Главный сервер занимает промежуточное положение между пользовательским интерфейсом и брокером сообщений: он принимает настройки, ведёт реестр источников и шаблонов, по расписанию создаёт задания и отслеживает их выполнение. Документы обрабатывают исполнительные узлы, получающие работу от брокера, поэтому отдельное соединение обработчика с пользовательским интерфейсом не требуется, а управление сбором сосредоточено на главном сервере.

На время выполнения задания исполнителю выдаются только необходимые ему параметры доступа. Выдачу обеспечивает доверенный контур – удостоверяющий центр и защищённое хранилище, где размещены сертификаты, ключи, маркеры доступа, учётные данные и параметры прокси-серверов. Журнал целостности решает иную задачу, подтверждая неизменность уже собранных документов, и с управлением доступом не связан.

Доставку заданий выполняет RabbitMQ с устойчивыми очередями и семантикой «не менее одного раза» (at-least-once). Сообщения публикуются как постоянные; подтверждение публикации означает их принятие брокером, а для постоянных сообщений в устойчивых очередях – сохранение на диске. Подтверждения используются на двух этапах: издатель получает подтверждение публикации от брокера, а потребитель подтверждает сообщение вручную после фиксации результата. При повторной доставке обработчик выполняет работу заново, не изменяя конечного состояния данных. В целевой архитектуре распределённый кэш (Redis) хранит короткоживущее состояние – ограничители частоты обращений, временные блокировки, сведения об аренде заданий и идентификаторы недавно обработанных публикаций; постоянное хранилище кэш не подменяет.

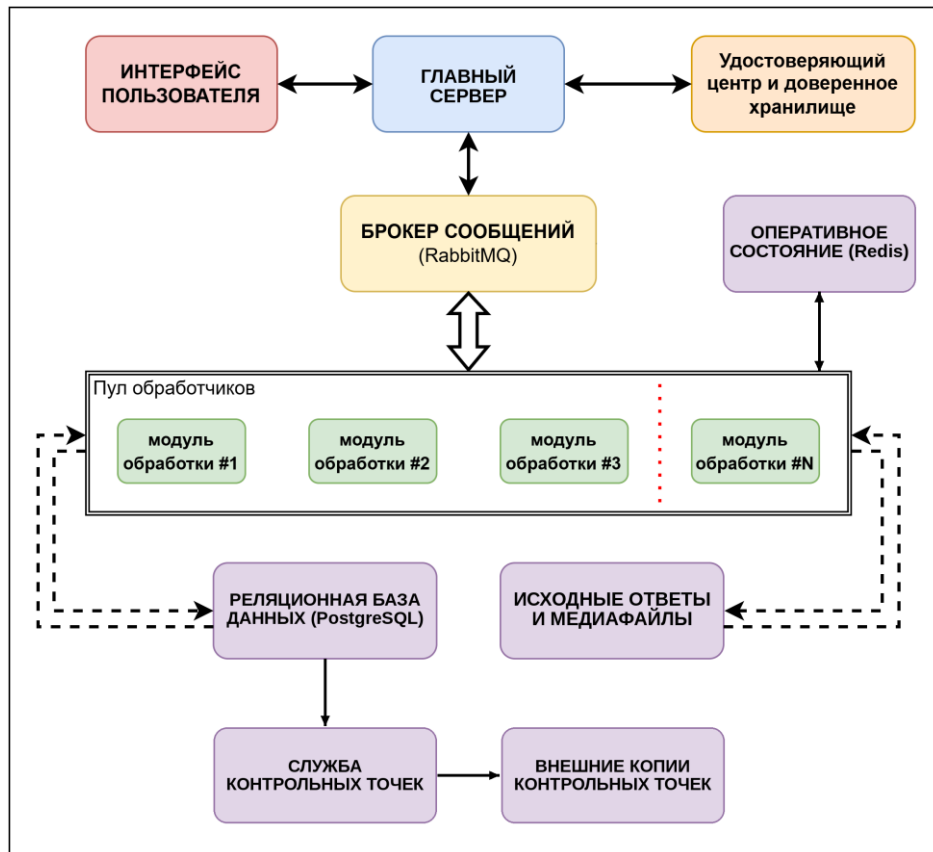


Рисунок 3 – Управление заданиями и хранение данных в целевой архитектуре приложения

Извлечённая публикация поступает в реляционную базу данных (PostgreSQL), а исходный ответ – на файловый сервер по протоколу SMB либо в совместимое объектное хранилище. В файловой части размещаются ответы форматов HTML, JSON, XML и PDF, а также медиафайлы; база хранит ссылки на объекты, их размеры и контрольные суммы. Помимо нормализованных публикаций в базе записываются источники, версии шаблонов, задания, попытки выполнения и сведения для контроля целостности; для полуструктурированных полей и служебных данных используется тип JSONB. На уровне корпуса идентичность публикации задаётся парой (*source_id*, *canonical_url*); хеш содержимого различает её состояния и не является самостоятельным ключом идентичности. На уровне выполнения отдельно задаётся ключ идемпотентности результата, сохраняемый при повторной доставке одного задания.

Загрузку по протоколу HTTP, браузерную обработку и локальный разбор выполняют узлы общей исполнительной группы. Эти узлы не обязаны быть одинаковыми: допускаются физические серверы и виртуальные машины с различным числом процессоров и объёмом памяти. Асинхронный режим применяется прежде всего к сетевому вводу-выводу, где велика доля времени ожидания ответа, а отдельные процессы используются для изоляции ошибок, размещения обработки на разных машинах и выполнения ресурсоёмких вычислений, к которым относятся распознавание текста на изображениях и преобразование медиафайлов.

Две частные схемы приведены на рисунке 4. На схеме а главный сервер направляет задания в несколько очередей, разделённых по типу обработчика, приоритету или группе источников. На схеме б модули чтения и записи выделены как логические компоненты доступа к хранилищу. Они могут входить в процесс исполнителя либо развёртываться отдельно; в измерительной реализации PostgreSQL доступен через клиентский модуль *asynpcrg* в каждом процессе.

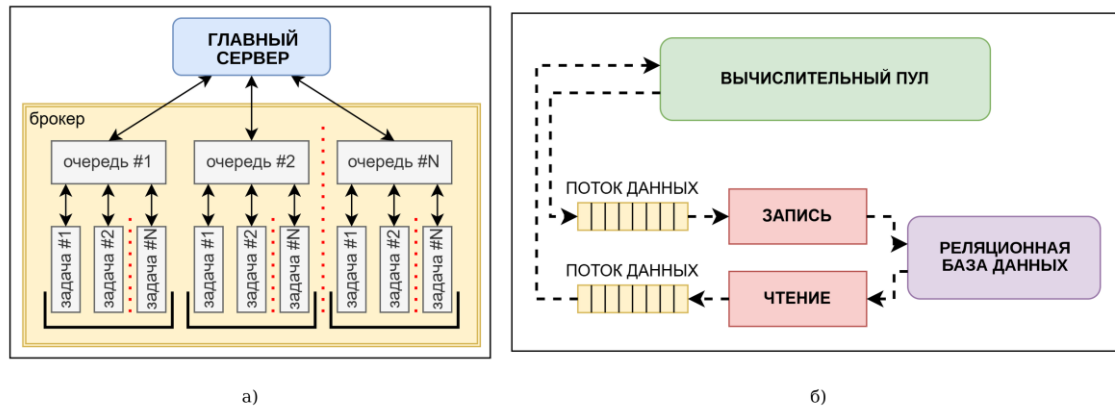


Рисунок 4 – Схемы работы с очередями (а) и с хранилищем данных (б); архитектурные принципы по [28]

Таблица 2 – Распределение операций и данных между компонентами

Компонент	Выполняемые операции	Состояние и передаваемые данные
Пользовательский интерфейс	Позволяет добавить источник, запустить задание, проследить выполнение и просмотреть ошибки	Настройки запуска и фильтры отображения
Главный сервер	Регистрирует шаблоны, ведёт расписание, маршрутизирует задания и контролирует выполнение	Версии конфигураций и состояние заданий
Доверенное хранилище	Выдаёт параметры доступа, сертификаты и ключи	Маркеры доступа, учётные данные и сертификаты
Брокер сообщений (RabbitMQ)	Сохраняет очереди, маршрутизирует задания по классу, обрабатывает подтверждения издателя и потребителя и повторную доставку	Постоянные сообщения, данные о попытках и служебные события
Распределённый кэш (Redis)	Ограничивает частоту обращений, ведёт учёт аренды заданий, устанавливает временные блокировки и проверяет недавние идентификаторы	Временные ключи, счётчики и блокировки с заданным сроком жизни
Исполнительные узлы	Загружают и проверяют документы, извлекают поля, выполняют предварительную обработку	Состояние текущей обработки на узле
Реляционная база данных (PostgreSQL)	В транзакциях фиксирует источники, шаблоны, задания, публикации и записи целостности	Реляционные данные с полями JSONB
Файловый сервер	Сохраняет исходные ответы и медиафайлы	Ответы HTML, JSON, XML, PDF; изображения, аудио, видео

Источник и версия шаблона закрепляются за заданием в момент его создания. Перед отправкой сообщения в соответствующую очередь планировщик присваивает заданию уникальный идентификатор и приоритет. Приняв сообщение, исполнитель получает документ от источника, сохраняет исходный ответ и, убедившись в выполнении условий conditionElement, извлекает поля по правилам выбранной версии шаблона.

Нормализованный результат передаётся модулю записи. Подтверждение брокеру отправляется после устойчивого сохранения результата либо явной фиксации окончательной ошибки. Неподтверждённое сообщение при отказе исполнителя возвращается в очередь и доставляется повторно; повторная запись обрабатывается по ключу идемпотентности.

Временные сбои и постоянные ошибки обрабатываются по-разному. Сетевой сбой, временная недоступность источника или превышение времени ожидания вызывают повторное обращение с увеличивающимся интервалом; отсутствие ресурса и другие постоянные ошибки фиксируются отдельно и повторов не вызывают. Несоответствие шаблону образует третий случай: ответ уже получен, поэтому документ сохраняют и направляют в очередь проверки конфигурации, а после исправления правил разбирают сохранённый ответ по новой версии шаблона.

Ключ идентичности публикации включает идентификатор источника и нормализованный адрес. До постановки задания в очередь схема и имя узла приводятся к каноническому виду, относительный путь

разрешается, фрагмент отбрасывается, а к служебным параметрам запроса применяются заданные в шаблоне правила отбора. Полученная пара используется при постановке заданий и проверке уникальности публикации. Очередной опрос или повторный разбор с другой версией шаблона не меняет эту предметную идентичность, но представляет собой новый запуск обработки, а не повторную доставку прежнего сообщения.

Повторная обработка ранее сохранённого файла начинается с передачи ссылки на него, а не самого файла: планировщик выбирает узел, имеющий быстрый доступ к объекту в хранилище. После разбора в базу данных возвращается нормализованный результат. Обмен через модули чтения и записи сокращает повторную передачу больших файлов и оставляет вычисления вблизи данных.

Для задания k в момент назначения t обозначим через $C_k(t)$ множество исполнителей, совместимых с требованиями к типу обработчика, памяти и доступу к данным. При непустом множестве допустимых узлов планировщик применяет эвристику МСТ [26] и выбирает исполнителя с наименьшей оценкой времени завершения согласно формуле (1):

$$i_{\text{sel}}(k, t) \in \underset{i \in C_k(t)}{\operatorname{argmin}} \left[Q_i(t) + \frac{c_k}{p_{i,g_k}} \right]. \quad (1)$$

Здесь $i_{\text{sel}}(k, t)$ – выбранный исполнитель; $Q_i(t) \geq 0$ – оценка времени завершения уже принятых узлом i заданий; $c_k > 0$ – трудоёмкость задания k , выраженная во времени обработки на эталонном узле; g_k – класс задания; $p_{i,g_k} > 0$ – безразмерный коэффициент производительности узла i для этого класса. Величины $Q_i(t)$ и $c_k/p_{i,g_k}$ выражаются в одной временной шкале, а сумма $t + Q_i(t) + c_k/p_{i,g_k}$ оценивает момент завершения. Общее слагаемое t на выбор узла не влияет.

В последовательной модели загрузка включает остаточное время текущего задания и сумму оценок для ожидающих заданий; при параллельной обработке требуется учитывать доступность исполнительской ёмкости узла. При совпадении минимальных оценок выбирается исполнитель с наименьшим идентификатором, а при отсутствии совместимых исполнителей задание остаётся в очереди. Ниже это правило сопоставлено с круговым распределением и с назначением наименее загруженному узлу.

Окна публикаций сопоставляются при каждом очередном опросе. При первом запуске лента считывается до заданной временной границы и формируется подтверждённое окно. Через интервал Δ запрашиваются последние N записей. Множество V_t объединяет идентификаторы всех страниц, прочитанных при текущем опросе, и сравнивается с предыдущим подтверждённым окном W_{t-1} . Условие достаточного пересечения записано в формуле (2).

$$|V_t \cap W_{t-1}| \geq O \quad (2)$$

Число общих публикаций должно быть не меньше порога O . В базовом режиме приняты значения $\Delta = 10$ минут, $N = 20$ и $O = 2$. Если пересечение оказывается меньше порога, читается следующая страница ленты либо расширяется окно последних записей. Обход завершается при выполнении условия (2), достижении временной границы или предельной глубины. До накопления в подтверждённом окне O записей применяется режим начального обхода.

Алгоритм 1 – Опрос ленты с проверкой перекрытия окон

Вход: S – источник; N – размер окна; O – порог пересечения; $D_{\max}$ – предельная глубина обхода.

Выход: U – множество новых публикаций; W_{cur} – подтверждённое окно.

1. $W_{\text{prev}} \leftarrow$ прочитать сохранённое подтверждённое окно для S .
2. $U \leftarrow \emptyset$; $V \leftarrow \emptyset$; $W_{\text{head}} \leftarrow \emptyset$; $\text{page} \leftarrow 1$; $\text{overlap} \leftarrow 0$.
3. ПОВТОРЯТЬ
4. $B \leftarrow$ прочитать страницу page ленты, начиная с последних N публикаций.
5. $B \leftarrow$ привести идентификаторы публикаций к нормализованному виду.
6. ЕСЛИ $\text{page} = 1$, ТО $W_{\text{head}} \leftarrow$ первые N публикаций B .
7. $U \leftarrow U \cup$ публикации из B , ещё не записанные в постоянное хранилище.
8. $V \leftarrow V \cup \text{ids}(B)$; $\text{overlap} \leftarrow |V \cap W_{\text{prev}}|$.
9. $\text{page} \leftarrow \text{page} + 1$.
10. ПОКА $\text{overlap} < O$ и $\text{page} \leq D_{\max}$ и временная граница не достигнута.
11. ЕСЛИ $\text{overlap} \geq O$, ТО
12. записать U ; $W_{\text{cur}} \leftarrow W_{\text{head}}$; сохранить W_{cur} для S .
13. ИНАЧЕ
14. сохранить U идемпотентно; зарегистрировать разрыв; поставить задание на расширенный обход.
15. КОНЕЦ ЕСЛИ



Рисунок 5 – Окна публикаций и их перекрытие при последовательных опросах. Зелёным обозначены вновь обнаруженные публикации, синим – ранее известные публикации в текущем окне, белым – публикации вне текущего окна

Пересечение окон служит признаком полноты только для лент с неизменными идентификаторами записей, доступной постраничной выдачей и устойчивым порядком от новых публикаций к старым: при таком порядке обход до ранее известной записи охватывает весь промежуток между опросами. Если источник добавляет старые материалы в середину ленты или переставляет записи, вывод о полноте по одному пересечению неправилен, поэтому для таких источников предусмотрены временной запас и периодический контрольный обход более глубокой части ленты.

4. Контроль целостности собранного корпуса

Хеш исходного ответа вычисляется до преобразования ответа в текст: из тела HTTP-ответа снимается транспортное сжатие, к полученной последовательности байтов применяется функция SHA-256 и лишь затем выполняются декодирование и нормализация. Из нормализованной публикации формируется объект JSON с фиксированным порядком полей: `source_id`, `canonical_url`, `captured_at`, `template_version`, `title`, `author`, `published_at`, `body`, `media`, `raw_sha256`. Строки приводятся к нормальной форме Unicode NFC, переводы строк – к виду LF, отсутствующее значение записывается как `null`, а временные отметки представляются по стандарту ISO 8601 в шкале UTC с микросекундной точностью и суффиксом Z.

Объект JSON записывается в кодировке UTF-8 без метки порядка байтов, необязательные пробелы не включаются, элементы массивов сохраняют исходный порядок, а ключи вложенных объектов упорядочиваются лексикографически. Именно эта последовательность байтов служит входом при вычислении хеша листа дерева по формуле (3) [25]:

$$\begin{aligned} \ell_i &= H(0x00 \| b_i), \\ M(D_n) &= H(0x01 \| M(D_{0:k}) \| M(D_{k:n})). \end{aligned} \quad (3)$$

В формуле (3) H обозначает SHA-256, b_i – каноническую последовательность байтов записи, а $\|$ – конкатенацию. Префикс `0x00` применяется к листьям, префикс `0x01` – к внутренним узлам. Для пустого списка $M(\emptyset) = H(\epsilon)$, для одной записи $M([b_i]) = H(0x00 \| b_i)$.

Дерево Меркла (рисунок 6) строится по упорядоченному списку канонических записей $D_n = (b_0, \dots, b_{n-1})$. Обозначения $D_{0:k}$ и $D_{k:n}$ соответствуют первой и второй частям списка. При $n > 1$ список разделяется после k -й записи, где k – наибольшая степень двойки, меньшая n , и корень вычисляется рекурсивно по формуле (3). Такой порядок охватывает любое число листьев без дублирования последнего узла [25]. Корневой хеш заносится в контрольную точку вместе с числом записей, номером пакета и временной отметкой UTC; контрольная точка сериализуется и подписывается по схеме Ed25519.

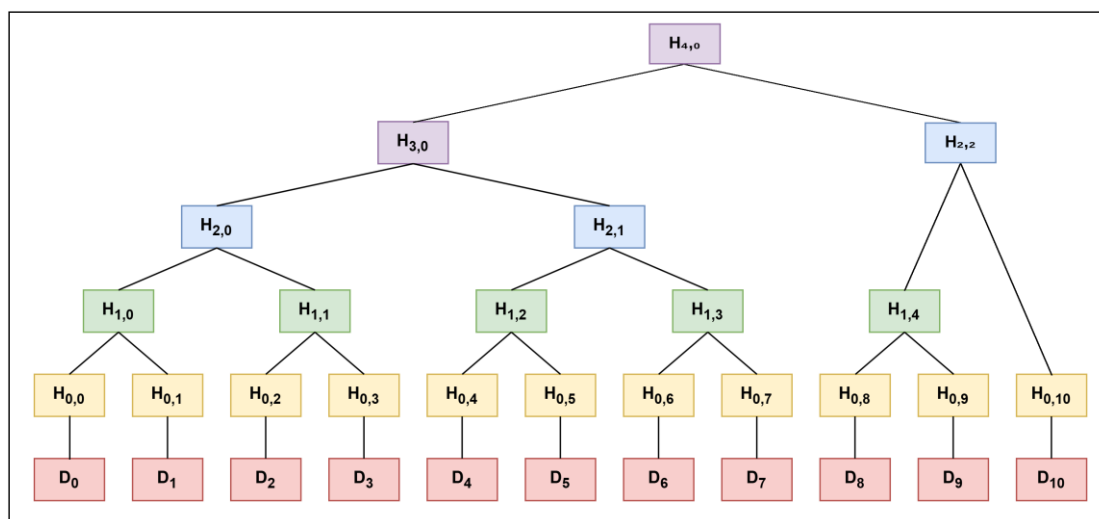


Рисунок 6 – Построение дерева Меркла по 11 каноническим записям корпуса; структура по [25]

Контрольная квитанция представляет собой вынесенную за основной контур копию подписанной контрольной точки, по которой сверяется ранее зафиксированное состояние корпуса. Подпись позволяет проверить контрольную точку по заранее доверенному открытому ключу, но сама по себе не доказывает, что предъявлена последняя зафиксированная версия корпуса. Внешняя квитанция закрепляет ожидаемые номер, объём и корневой хеш пакета и позволяет обнаружить откат относительно этого состояния, в том числе подмену более ранней корректно подписанной точкой.

Модель угроз предполагает, что нарушитель может изменять и удалять документы, переставлять записи и откатывать базу данных, но не контролирует службу контрольных точек и независимо сохранённые квитанции. Доверенными считаются первоначальная фиксация, служба подписания и заранее известный проверяющему открытый ключ; закрытый ключ Ed25519 должен оставаться в контуре службы и быть недоступным обработчикам и учётной записи базы данных. Компрометация этой службы или ключа, а также захват административного доступа к общему физическому серверу в данную модель не входят. Поэтому изоляция контейнеров на стенде рассматривается как логическое разделение, а не защита от компрометации всего узла.

Принадлежность публикации пакету из 128 записей проверяется по канонической записи, её индексу, числу листьев, семи соседним хешам и подписанной контрольной точке. Проверяющий вычисляет хеш листа, последовательно объединяет его с хешами пути и сравнивает результат со значением в контрольной точке. Объём пути растёт логарифмически с размером пакета. Доказательство согласованности двух размеров дерева проверяет сохранение его прежнего префикса при добавлении записей [25].

5. Методика экспериментальной оценки

Экспериментальная проверка выполнена на экспериментальной реализации описанной архитектуры, в котором роли компонентов закреплены за следующими программными средствами: транзакционное хранение – PostgreSQL 17.11, распределённый кэш и координация короткоживущего состояния – Redis 8.10.1, доставка заданий – RabbitMQ 4.0.9.

Обработчики реализованы на языке Python версии 3.12.14; сценарии управления стендом запускались также в Python 3.14.4. Использованы библиотеки aiohttp 3.14.3, aio-pika 9.6.2, asyncpg 0.31.0 и lxml 6.1.3. Завершение транзакции в базе данных предшествовало подтверждению сообщения брокеру. В испытании идемпотентность фиксации обеспечивалась составным уникальным ключом (run_id, article_id), где run_id обозначает отдельный прогон, а article_id – документ тестового корпуса. Этот ключ исключает повторную запись документа внутри одного прогона и не заменяет идентификацию публикации по (source_id, canonical_url) между периодическими опросами.

После компонентных испытаний проведены дополнительные серии: сравнение со Scrapy 2.18.0 при одинаковом числе процессов, работа с общим ограничителем запросов, раздельное включение инфраструктурных компонентов, отказные эпизоды, повторный разбор сохранённых HTML и измерение операций дерева Меркла. В основной серии масштабирования использованы один, два, четыре и восемь исполнителей с десятью повторами на каждую конфигурацию. Расширенная точка серии сохранена в приложении А.

Стенд был одноузловым: контейнеры выполнялись в одной виртуальной машине под управлением Ubuntu с 48 логическими процессорами AMD EPYC-Milan и доступной оперативной памятью около 47 ГиБ. Параметры

приведены в таблице 3. Оба решения получали одинаковые детерминированные HTML и правила XPath; для загрузки использованы восемь одинаковых реплик источника с заданной задержкой ответа 18–26 мс.

Таблица 3 – Конфигурация дополнительных испытаний

Компонент	Назначение	Параметры дополнительных испытаний
PostgreSQL	Фиксация результатов	17.11; synchronous_commit=on; ключ (run_id, article_id)
RabbitMQ	Доставка заданий	4.0.9; постоянные сообщения; подтверждения; до 8 очередей
Redis	Оперативное состояние	8.10.1; в scaling – асинхронная телеметрия; в абляции – синхронный счётчик
Приложение / Scrapy	Сравнение конфигураций	Python 3.12.14; Scrapy 2.18.0; 1, 2, 4, 8 процессов; 25 одновременных запросов на процесс
Пулы PostgreSQL	Доступ к базе	От 1 до 8 соединений на процесс в основном диапазоне; одинаково для обоих решений
Нагрузка	Синтетическая серия	5000 HTML на прогон; 8 реплик источника; задержка 18–26 мс; lxml и одинаковый XPath
Повторы	Разброс результатов	10 на конфигурацию scaling; 10 на вариант абляции; 5 на уровень общего ограничителя
Среда	Один узел	Ubuntu; Docker 29.1.3; Compose 2.40.3; 48 vCPU; около 47 ГиБ RAM
Отдельные блоки	Дополнительная проверка	Три единичных отказных эпизода; 16 реальных HTML; деревья от 10^3 до 10^6 записей

Для каждого прогона измерялись длительность обработки, пропускная способность и 95-й процентиль времени от общего начала пакета до SQL-отметки документа. Отдельный коллектор завершал отсчёт после появления всех уникальных результатов в базе данных. В SQL сохранялась отметка clock_timestamp(), соответствующая вычислению выражения внутри команды [30]. Подтверждение завершённого COMMIT и получение АСК выполняются позднее и в эту отметку не входят. Данные CPU снимались отдельно для прикладных процессов и инфраструктурных сервисов.

Пропускная способность определена как отношение числа корректно сохранённых документов к длительности измеряемого прогона (формула (4)):

$$P = \frac{N_{ok}}{T}. \quad (4)$$

В формуле (4) P – пропускная способность, N_{ok} – число уникальных результатов, видимых коллектору, $T = t_{seen} - t_0$ – интервал от заданного начала пакета до обнаружения его завершения. Период опроса коллектора составляет 10 мс. В серии масштабирования предварительное заполнение очередей завершалось до назначения t_0 ; общая отметка задавалась на 2,5 с вперёд, после чего запускались контейнеры. Барьер готовности не применялся.

Сравниваемые реализации использовали lxml и asynpcrg, одинаковую степень параллелизма и одинаковые пределы пулов соединений при каждом числе процессов. Восемь реплик источника и до восьми очередей RabbitMQ использовались во всей новой серии. В масштабируемом варианте Redis обслуживал асинхронную телеметрию. Синхронный счётчик в базовом варианте, ресурсная серия и абляция рассматриваются отдельно.

Ускорение и эффективность рассчитывались по средним пропускным способностям: $S(n) = \bar{P}_n / \bar{P}_1$, $E(n) = S(n)/n$. В новых таблицах приведены средние и 95%-е доверительные интервалы $\bar{x} \pm t_{0,975,r-1}s/\sqrt{r}$, где r – число прогонов, s – выборочное стандартное отклонение. Для задержки усреднялись внутрипакетные 95-е проценты. В старых компонентных таблицах 4–6 сохранены результаты исходной серии с указанной для неё методикой.

6. Результаты экспериментальной оценки

Функциональная проверка подтвердила возможность повторного разбора сохранённого исходного ответа после изменения версии шаблона без повторного обращения к источнику. Дополнительная проверка на HTML действующих новостных ресурсов и показатели совпадения извлечённых полей приведены в таблице 11.

Компонентное испытание сетевого сборщика и обработчика правил выполнено на 600 HTML-документах без участия брокера сообщений, распределённого кэша и базы данных; его результаты приведены в таблице 4.

Таблица 4 – Компонентное испытание загрузки и разбора HTML-документов

Режим	Время, с	Пропускная способность, док./с	Корректно извлечено
Последовательный режим	$8,350 \pm 0,129$	$71,9 \pm 1,1$	600 из 600
Асинхронный режим, 8 соединений	$1,031 \pm 0,032$	$582,5 \pm 17,9$	600 из 600
Асинхронный режим, 16 соединений	$0,556 \pm 0,082$	$1094,1 \pm 140,5$	600 из 600
Асинхронный режим, 32 соединения	$0,291 \pm 0,008$	$2062,5 \pm 54,9$	600 из 600

Переход от последовательного режима к восьми одновременным соединениям увеличил пропускную способность в 8,1 раза, а при 32 соединениях достигнуто 2062,5 документа в секунду при полном извлечении полей во всех прогонах. Полученные значения обосновывают асинхронную организацию сетевого ввода-вывода в исполнительных узлах.

Имитация распределения заданий выполнена для 100 независимых наборов с начальными состояниями генератора от 1 до 100, каждый из которых содержал 500 заданий. Базовая трудоёмкость задания формировалась как экспонента нормально распределённой величины со средним 0 и стандартным отклонением 0,55; с вероятностью 0,20 задание относилось к ресурсоёмкому классу. В модели использовались четыре исполнителя. Каждому задавалась пара коэффициентов производительности для обычных и ресурсоёмких заданий: (1,0; 1,0), (1,3; 0,9), (0,8; 0,6) и (1,7; 1,8) соответственно. В начале обработки каждого набора очереди всех исполнителей были пустыми. Длительность задания рассчитывалась делением его трудоёмкости на коэффициент производительности выбранного узла для соответствующего класса. Правила назначения сравнивались по времени завершения последнего задания в наборе.

Для повторного расчёта использовался генератор `random.Random(seed)` в Python. При формировании каждого задания сначала вызывался метод `gauss`, задающий нормально распределённую величину, затем метод `random`, определяющий класс задания. Средняя длительность составила 194,178 для кругового назначения, 125,884 для выбора наименее загруженного исполнителя и 123,279 для МСТ. Эти значения относятся к повторному расчёту; в таблице 5 приведены результаты исходной серии. В скрипте воспроизведения закреплены используемый генератор и последовательность его вызовов.

Таблица 5 – Длительность выполнения набора заданий при разных правилах назначения

Правило распределения заданий	Средняя длительность выполнения набора	Сокращение длительности относительно кругового назначения
Круговое назначение	$194,7 \pm 11,6$	База сравнения
Выбор исполнителя с наименьшей загрузкой	$125,9 \pm 3,8$	35,3 %
Выбор по ожидаемому моменту завершения — МСТ, формула (1)	$123,1 \pm 3,5$	36,8 %

В исходной серии минимальная средняя длительность получена для правила (1): на 36,8 % меньше, чем при круговом назначении, и на 2,2 % меньше, чем при выборе только по текущей загрузке. Основное сокращение времени связано с учётом накопленной очереди. Дополнительный учёт производительности узлов уменьшает длительность выполнения набора в рассмотренной неоднородной группе исполнителей.

Поступление публикаций в течение 24 часов моделировалось неоднородным пуассоновским процессом. Рассмотрены четыре базовые интенсивности: 0,5; 1,0; 2,0 и 3,0 публикации в минуту. На интервале с 12:00 до 16:00 каждое из этих значений увеличивалось вдвое, имитируя дневной рост активности источника. Опрос выполнялся каждые 10 минут, размер страницы и окна составлял 20 записей, порог пересечения – 2 записи, предельная глубина – 10 страниц. Для каждого уровня использованы 100 начальных состояний генератора 2026090600–2026090699. Фиксированный вариант читал одну страницу, адаптивный – последующие страницы до достижения порога пересечения или предельной глубины. Кратность чтения определялась отношением числа полученных записей с повторами к числу уникально собранных публикаций.

Таблица 6 – Имитационная оценка перекрывающегося опроса при размере окна 20 записей и шаге 10 минут

Базовая интенсивность, публ./мин	Публикаций за сутки	Пропуски фиксированного окна, %	Пропуски адаптивного обхода, %	Кратность чтения	Среднее число страниц	95-й процентиль глубины, страниц
0,5	839	0,00	0,00	3,40	1,00	1
1,0	1680	2,42	0,00	1,89	1,11	2
2,0	3360	20,57	0,00	1,52	1,78	3
3,0	5038	42,96	0,00	1,31	2,30	4

Имитационный эксперимент включает по 100 повторений для четырёх уровней интенсивности и двух алгоритмов, всего 800 расчётных вариантов. При базовой интенсивности 0,5 публикации в минуту фиксированное окно практически не теряло записей, создавая при этом избыточное чтение с кратностью 3,40. С ростом интенсивности до 2 и 3 публикаций в минуту доля пропущенных фиксированным окном публикаций увеличивалась до 20,57 и 42,96 % соответственно, тогда как адаптивный обход пропусков не допустил и увеличивал глубину чтения только при недостаточном пересечении окон.

Нулевое число пропусков получено для принятой модели источника с устойчивой обратной хронологией, постоянными идентификаторами записей и доступной постраничной выдачей; ограничения этой модели отмечены в разделе 3.

Вычислительные затраты дерева Меркла и проверочных путей исследованы для четырёх размеров корпуса (таблица 7). На каждом размере проверены 100 путей включения и 100 доказательств согласованности между половинным и полным размером дерева. Таймер построения включает генерацию тестовых записей, сериализацию, хеширование и сборку дерева.

Таблица 7 – Построение дерева Меркла и проверка включения записи

Записей	Построение, мс	RSS после сборки, МБ	Путь включения, байт (среднее)	Проверка пути, мкс
1 000	5,66	33,1	320,00	13,75
10 000	55,20	35,9	436,48	14,83
100 000	561,37	51,1	541,44	19,39
1 000 000	5378,53	203,0	640,00	24,50

Для миллиона записей построение заняло 5,379 с, средний путь включения – 640 байт, его проверка – 24,50 мкс. Доказательство согласованности между 500 000 и 1 000 000 записей занимало 512 байт и проверялось за 13,52 мкс. Все сформированные проверочные пути прошли верификацию; построение каждого размера измерено однократно, а времена операций проверки усреднены по 100 выполненным операциям.

Результаты основной серии масштабирования приведены в таблице 8. Для каждого числа процессов выполнены десять повторов каждой системы; все прогоны завершились сохранением полного тестового набора. Порядок запуска решений и направления обхода точек параллелизма чередовались.

Таблица 8 – Пакетная производительность при одинаковом числе процессов; среднее $\pm$ 95%-й доверительный интервал

Число процессов	Приложение, док./с	Scrapy, док./с	Приложение: p95 до SQL, мс	Scrapy: p95 до SQL, мс
1	930,3 $\pm$ 4,1	464,5 $\pm$ 12,3	5108,7 $\pm$ 23,3	10457,3 $\pm$ 282,3
2	1775,4 $\pm$ 10,6	910,9 $\pm$ 9,7	2663,8 $\pm$ 14,8	5154,9 $\pm$ 93,1
4	3255,0 $\pm$ 32,9	1707,6 $\pm$ 35,4	1447,7 $\pm$ 13,0	2745,3 $\pm$ 38,3
8	5204,2 $\pm$ 120,9	3066,8 $\pm$ 130,9	891,2 $\pm$ 17,6	1487,1 $\pm$ 66,9

В диапазоне от одного до восьми исполнителей отношение средних пропускных способностей приложения и Scrapy составило соответственно 2,00; 1,95; 1,91 и 1,70. Средняя пакетная задержка до SQL-отметки также была ниже у приложения. Эффективность относительно однопроцессного режима при восьми исполнителях составила 0,699 для приложения и 0,825 для Scrapy. Дополнительная точка и границы диапазона приведены в приложении А.

Отдельная серия с четырьмя процессами включала CPU-затраты RabbitMQ и Redis в границу приложения (таблица 13). Эти показатели относятся к базовой реализации и дополняют сравнение пропускной способности; их временной интервал охватывает также часть запуска и завершения компонентов.

Таблица 9 – Результаты трёх отказных эпизодов

Сценарий	Сохранено / уникальных	Повторные доставки	Счётчик processed	Устойчивые дубликаты
Завершение исполнителя	3000 / 3000	25	3000	0
Перезапуск RabbitMQ	3000 / 3000	1902	3004	0
Перезапуск Redis	3000 / 3000	33	3001	0

При принудительном завершении одного исполнителя брокер повторно доставил 25 сообщений. В итоговой таблице сохранились все 3000 уникальных документов. Проверка относится к одному отказному эпизоду с установленной предварительной выдачей 25 сообщений.

Перезапуск RabbitMQ сопровождался 1902 повторными доставками, а перезапуск Redis – 33. В каждом случае итоговый набор содержал 3000 уникальных документов. Счётчик processed зарегистрировал 3004 и 3001 события соответственно; он отражает прохождение точки инструментирования, а число попыток транзакционной записи отдельно не измерялось.

Таблица 10 – Подписание и проверка контрольных точек Ed25519

Число записей	Данные точки, байт	Подпись, байт	Подписание, мкс	Проверка подписи, мкс
1 000	179	64	35,97	113,46
10 000	180	64	36,71	114,59
100 000	181	64	36,80	118,61
1 000 000	182	64	36,24	115,39

На каждом размере дерева выполнено 500 операций подписания и проверки контрольной точки. Для миллиона записей среднее время подписи составило 36,24 мкс, проверки – 115,39 мкс; размер подписи – 64 байта. Измерение проведено локально и характеризует криптографические операции над подготовленной контрольной точкой.

В выборку вошли по восемь страниц Guardian Technology и TechCrunch. Исходные ответы сохранены; в локальной копии DOM заменены структурные элементы заголовка и статьи. Затем к копии применены прежний и исправленный шаблоны. Непустые исходные поля использованы как контроль для точного сравнения после повторного разбора.

Таблица 11 – Повторный разбор сохранённых HTML после управляемого изменения структуры

Поле	Исходный v1	После изменения: v1	После изменения: v2	Точное совпадение / сравниваемых
Заголовок	16/16	0/16	16/16	16/16
Автор	16/16	16/16	16/16	16/16
Дата	16/16	16/16	16/16	16/16
Основной текст	8/16	0/16	16/16	8/8

Исправленный шаблон обеспечил заполнение всех четырёх полей на 16 страницах. Повторный разбор выполнялся локально; счётчик новых HTTP-запросов на этом этапе равнялся нулю. Контроль представляет восстановление значений исходного шаблона, а точность выделения основного текста относительно экспертной разметки отдельно не оценивалась.

В контрольной серии оба решения обращались к одному прокси-ограничителю. Для каждого уровня выполнены пять повторов с четырьмя процессами и пакетом из 1000 документов. Внутренние ограничители были отключены; ограничение задавалось для общего входного потока.

Таблица 12 – Производительность с единым внешним ограничителем запросов

Предел, запросов/с	Приложение, док./с	Scrapy, док./с	Приложение: p95 до SQL, мс	Scrapy: p95 до SQL, мс
50	47,83	48,05	19 855,87	19 767,24
100	93,43	93,38	10 159,00	10 168,50
150	137,07	138,15	6925,98	6869,68
200	175,62	176,48	5406,36	5374,41

При общем ограничителе различие средних пропускных способностей не превышало 1 %. Заданный предел и фактически достигнутая интенсивность приведены отдельно: реализация ограничителя включает ожидания планировщика. Эту серию следует рассматривать как характеристику работы при ограниченном входном потоке.

Таблица 13 – Средние CPU-затраты базовой реализации; пять повторов, четыре процесса

Система	Процессы, CPU·с	PostgreSQL, CPU·с	RabbitMQ, CPU·с	Redis, CPU·с	Сумма, CPU·с
Приложение	5,601	1,687	4,894	0,275	12,457
Scrapy	19,094	2,129	—	—	21,223

Данные таблицы 13 восстановлены из сохранённой сводки ранней серии. Учитывались CPU-счётчики cgroup прикладных контейнеров и указанных сервисов. Генератор источника не включён в сравниваемый контур; таймеры охватывали также часть запуска и завершения процессов. Для памяти сохранены отдельные максимумы контейнеров, поэтому общий синхронный максимум по этой серии не вычислялся.

В варианте A0 выполнялись прямые HTTP-запросы, разбор и одиночная запись в базу. A1 добавлял доставку через устойчивую очередь; A2 – синхронное обновление счётчика Redis; A3 – канонизацию, хеширование листа и атомарную запись результата вместе с листом. Корень пакета строился отдельно по подготовленным хешам.

Таблица 14 – Последовательное включение компонентов; среднее $\pm$ 95%-й доверительный интервал, десять повторов

Код	Состав	Пропускная способность, док./с	p95 до SQL, мс
A0	HTTP → разбор → PostgreSQL	3479,72 $\pm$ 53,30	1356,62 $\pm$ 21,06
A1	Очередь RabbitMQ и исполнители	1200,95 $\pm$ 45,03	4099,43 $\pm$ 161,18
A2	A1 и счётчик Redis	1188,39 $\pm$ 25,55	4135,67 $\pm$ 90,81
A3	A2, хеш листа и атомарная запись	1135,91 $\pm$ 24,26	4321,86 $\pm$ 93,56

В абляционном сценарии временная отметка задавалась до предварительной публикации сообщений для A1–A3, тогда как A0 обходился без публикации. Поэтому разница A0–A1 характеризует варианты запуска в целом. Для A2–A3 среднее парное относительное изменение пропускной способности составило $-4,37$ % с 95%-м интервалом $\pm 2,18$ процентного пункта. Эта оценка включает сериализацию, хеширование, вторую запись и явную транзакцию; подписание контрольной точки измерялось в отдельном микробенчмарке.

Свёртка 5000 заранее вычисленных листовых хешей в Merkle-корень заняла в среднем $4,390 \pm 0,814$ мс. Чтение листьев из базы данных выполнено до запуска этого таймера. Доверительный интервал среднего рассчитан по десяти повторам.

7. Обсуждение и ограничения применимости

Результаты испытаний согласуются с заложенным порядком обработки: повторная доставка заданий сопровождается идемпотентной записью, а сохранённые исходные ответы позволяют менять правила извлечения независимо от загрузки. В основной серии на 1–8 процессах приложение показало более высокую пакетную производительность, чем выбранная конфигурация Scrapy. Общий ограничитель входного потока сближает показатели решений (таблица 12). Абляция отдельно показывает затраты добавляемых компонентов (таблица 14).

Область применимости полученных оценок ограничена условиями эксперимента. Испытания выполнены в одной виртуальной машине; задержки между физическими серверами не исследованы. Синтетические документы обеспечивают воспроизводимую нагрузку, а сохранённые HTML действующих источников

используются для проверки шаблонов. Конечный размер пакета и запуск без барьера готовности влияют на измеряемую производительность. Сопоставление характеризует исследованные конфигурации; расширение диапазона приводит к сближению кривых (приложение А). Выводы о полноте опроса относятся к лентам с устойчивым обратно-хронологическим порядком. Журнал удостоверяет состояние корпуса относительно контрольной точки; подлинность публикации и защита ключа требуют самостоятельных мер.

Заключение

Разработаны архитектура и экспериментальная реализация приложения для автоматизированного сбора публикаций из открытых источников массовой информации. Компоненты разделены по этапам обработки: доставку заданий обеспечивает брокер сообщений с устойчивыми очередями, обработку – группа исполнительных узлов, фиксацию результатов – реляционная база данных; распределённый кэш используется для короткоживущего состояния и постоянное хранилище не заменяет. Целевая схема предусматривает обращение исполнителей к файловому хранилищу и получение параметров доступа из доверенного контура, а повторная доставка заданий предусмотрена порядком обработки и не нарушает целостности результатов.

Подключение источника выполняется изменением конфигурационного шаблона, а не программного ядра. В шаблоне перечислены признаки применимости, исключаемые области и извлекаемые поля, для поиска которых заданы выражения XPath, CSS-селекторы и регулярные выражения; там же указаны расписание сбора и параметры сохранения медиафайлов. Хранение исходного ответа вместе с версией шаблона позволяет повторно разобран документ после исправления правил, не запрашивая его у источника.

Опрос ленты начинается с заданного числа последних публикаций и продолжается вглубь, пока число записей, общих с подтверждённым окном, не достигнет порога. В имитационной серии такой обход не оставил пропусков во всём исследованном диапазоне интенсивности при устойчивой постраничной выдаче, причём дополнительная глубина использовалась только при недостаточности начального окна.

Состояние собранных документов фиксируется с помощью упорядоченного дерева Меркла, подписанных контрольных точек и внешних копий контрольных значений. Испытания операций дерева и подписи показали компактность проверочных путей и невысокую стоимость их обработки: для миллиона записей путь включения занимает в среднем 640 байт и проверяется за 24,50 мкс.

В основной серии от одного до восьми исполнителей средняя пропускная способность приложения увеличилась с 930,26 до 5204,17 документа в секунду; отношение к результату выбранной конфигурации Scrapy составило 1,70–2,00. Повторный разбор сохранённых HTML восстановил ранее извлекаемые поля после управляемой замены структуры без новых сетевых обращений. Добавление канонизации, хеша листа и атомарного сохранения двух связанных записей изменило пропускную способность абляционного варианта в среднем на –4,37 %.

В трёх отказных эпизодах с завершением исполнителя и перезапуском RabbitMQ и Redis сохранены все уникальные документы тестового набора. Дальнейшая работа связана с многосерверным размещением компонентов, единым барьером начала измерений, физическим разделением основного контура и службы контрольных точек, а также с длительным наблюдением действующих источников.

Список литературы

1. Kahlon N.K., Singh W. A systematic review of web scraping: Techniques, LLM-enhanced approaches, performance metrics, and legal-ethical issues. *Data & Knowledge Engineering*. 2026;164:102598. DOI: 10.1016/j.datak.2026.102598.
2. Landeta-López P., García J.M., Guevara-Vega C., Ruiz-Cortés A. LLMs applied to web scraping and web crawling: a systematic review. *Computing*. 2026;108:78. DOI: 10.1007/s00607-026-01666-5.
3. Ferrara E., De Meo P., Fiumara G., Baumgartner R. Web data extraction, applications and techniques: A survey. *Knowledge-Based Systems*. 2014;70:301–323. DOI: 10.1016/j.knosys.2014.07.007.
4. Uzun E. A Novel Web Scraping Approach Using the Additional Information Obtained From Web Pages. *IEEE Access*. 2020;8:61726–61740. DOI: 10.1109/ACCESS.2020.2984503.
5. Patnaik S.K., Babu C.N., Bhawe M. Intelligent and Adaptive Web Data Extraction System Using Convolutional and Long Short-Term Memory Deep Learning Networks. *Big Data Mining and Analytics*. 2021;4(4):279–297. DOI: 10.26599/BDMA.2021.9020012.
6. Дубовицкий С.С., Бедрин П.А., Яцков А.К., Варламов М.И. Автоматическое построение правил извлечения информации для новостных веб-сайтов. *Труды ИСП РАН*. 2024;36(5):153–162. DOI: 10.15514/ISPRAS-2024-36(5)-11.

7. Эйзенах Д.С., Рамейков А.С., Никифоров И.В. Высокопроизводительный распределенный веб-скрапер. Труды ИСП РАН. 2021;33(3):87–100. DOI: 10.15514/ISPRAS-2021-33(3)-7.
8. Лазарев В.А., Варламов М.И., Яцков А.К. Система управления заданиями автоматизированного сбора данных из сети Интернет. Труды ИСП РАН. 2022;34(2):111–122. DOI: 10.15514/ISPRAS-2022-34(2)-9.
9. Левашкин С.П., Иванов К.Н., Кушуков С.В. Ферма данных: информационная система сбора, хранения и обработки неструктурированных данных из разнородных источников. Труды ИСП РАН. 2023;35(2):57–72. DOI: 10.15514/ISPRAS-2023-35(2)-5.
10. Серенко Д.С., Терентьев Е.Д., Зубарев Д.В., Соченков И.В. Архитектура системы сбора и извлечения информации для интеллектуальной поисково-аналитической системы. Труды ИСП РАН. 2025;37(2):263–280. DOI: 10.15514/ISPRAS-2025-37(2)-20.
11. Лапкина А.В., Петухов А.А. Классификация HTTP-запросов к серверу в задаче автоматического обхода современных веб-приложений. Труды ИСП РАН. 2021;33(3):77–86. DOI: 10.15514/ISPRAS-2021-33(3)-6.
12. Tacuri A., Firmenich S., Fernández A., Riva F., Urbieto M., Rossi G. Web Scraping by End Users. IEEE Access. 2025;13:205027–205044. DOI: 10.1109/ACCESS.2025.3636662.
13. Waseem M., Liang P., Shahin M. A Systematic Mapping Study on Microservices Architecture in DevOps. Journal of Systems and Software. 2020;170:110798. DOI: 10.1016/j.jss.2020.110798.
14. Zhou X., Li S., Cao L., Zhang H., Jia Z., Zhong C., et al. Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. Journal of Systems and Software. 2023;195:111521. DOI: 10.1016/j.jss.2022.111521.
15. Петушков Г.В., Сигов А.С. Анализ и выбор структуры многопроцессорной вычислительной системы по критерию быстродействия. Russian Technological Journal. 2024;12(6):20–25. DOI: 10.32362/2500-316X-2024-12-6-20-25.
16. Giamattei L., Guerriero A., Pietrantuono R., Russo S., Malavolta I., Islam T., et al. Monitoring tools for DevOps and microservices: A systematic grey literature review. Journal of Systems and Software. 2024;208:111906. DOI: 10.1016/j.jss.2023.111906.
17. Alexandrescu A., Butincu C.N. Decentralized News-Retrieval Architecture Using Blockchain Technology. Mathematics. 2023;11(21):4542. DOI: 10.3390/math11214542.
18. Pan B., Stakhanova N., Ray S. Data Provenance in Security and Privacy. ACM Computing Surveys. 2023;55(14s):323. DOI: 10.1145/3593294.
19. Soriano-Salvador E., Guardiola-Múzquiz G. SealFS: Storage-based tamper-evident logging. Computers & Security. 2021;108:102325. DOI: 10.1016/j.cose.2021.102325.
20. Guardiola-Múzquiz G., Soriano-Salvador E. SealFSv2: combining storage-based and ratcheting for tamper-evident logging. International Journal of Information Security. 2023;22:447–466. DOI: 10.1007/s10207-022-00643-1.
21. Laaber C., Scheuner J., Leitner P. Software microbenchmarking in the cloud. How bad is it really? Empirical Software Engineering. 2019;24:2469–2508. DOI: 10.1007/s10664-019-09681-1.
22. Scrapy-Redis. Redis-based components for Scrapy. Documentation. URL: <https://scrapy-redis.readthedocs.io/en/master/readme.html> (дата обращения: 27.09.2026).
23. IST Research. Scrapy Cluster: distributed crawling architecture and implementation. URL: <https://github.com/istresearch/scrapy-cluster> (дата обращения: 27.09.2026).
24. ISO 28500:2017. Information and documentation — WARC file format. Geneva: International Organization for Standardization; 2017. URL: <https://www.iso.org/standard/68004.html> (дата обращения: 27.09.2026).
25. Laurie B., Messeri E., Stradling R. Certificate Transparency Version 2.0. RFC 9162. 2021. DOI: 10.17487/RFC9162.
26. Braun T.D., Siegel H.J., Beck N., Bölöni L.L., Maheswaran M., Reuther A.I., Robertson J.P., Theys M.D., Yao B., Hensgen D., Freund R.F. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. Journal of Parallel and Distributed Computing. 2001;61(6):810–837. DOI: 10.1006/jpdc.2000.1714.
27. Cho J., Garcia-Molina H. Effective page refresh policies for Web crawlers. ACM Transactions on Database Systems. 2003;28(4):390–426. DOI: 10.1145/958942.958945.
28. Richards M., Ford N. Fundamentals of Software Architecture: An Engineering Approach. Sebastopol: O'Reilly Media; 2020. 432 p. ISBN: 9781492043447.
29. Koster M., Illyes G., Zeller H., Sassman L. Robots Exclusion Protocol. RFC 9309. 2022. DOI: 10.17487/RFC9309.
30. PostgreSQL Global Development Group. PostgreSQL 17 Documentation. Date/Time Functions and Operators. URL: <https://www.postgresql.org/docs/17/functions-datetime.html> (дата обращения: 27.09.2026).

31. YAML Language Development Team. YAML Ain't Markup Language (YAML™), version 1.2.2. 2021. URL: <https://yaml.org/spec/1.2.2/> (дата обращения: 27.09.2026).

References

1. Kahlon N.K., Singh W. A systematic review of web scraping: Techniques, LLM-enhanced approaches, performance metrics, and legal-ethical issues. *Data & Knowledge Engineering*. 2026;164:102598. DOI: 10.1016/j.datak.2026.102598.
2. Landeta-López P., García J.M., Guevara-Vega C., Ruiz-Cortés A. LLMs applied to web scraping and web crawling: a systematic review. *Computing*. 2026;108:78. DOI: 10.1007/s00607-026-01666-5.
3. Ferrara E., De Meo P., Fiumara G., Baumgartner R. Web data extraction, applications and techniques: A survey. *Knowledge-Based Systems*. 2014;70:301–323. DOI: 10.1016/j.knosys.2014.07.007.
4. Uzun E. A Novel Web Scraping Approach Using the Additional Information Obtained From Web Pages. *IEEE Access*. 2020;8:61726–61740. DOI: 10.1109/ACCESS.2020.2984503.
5. Patnaik S.K., Babu C.N., Bhawe M. Intelligent and Adaptive Web Data Extraction System Using Convolutional and Long Short-Term Memory Deep Learning Networks. *Big Data Mining and Analytics*. 2021;4(4):279–297. DOI: 10.26599/BDMA.2021.9020012.
6. Dubovitskii S.S., Bedrin P.A., Yatskov A.K., Varlamov M.I. Avtomaticheskoe postroenie pravil izvlecheniya informatsii dlya novostnykh veb-saitov [Automatic construction of information extraction rules for news websites]. *Trudy ISP RAN = Proceedings of ISP RAS*. 2024;36(5):153–162. (In Russ.). DOI: 10.15514/ISPRAS-2024-36(5)-11.
7. Eizenakh D.S., Rameikov A.S., Nikiforov I.V. Vysokoproizvoditel'nyi raspredelennyi veb-skraper [High-performance distributed web scraper]. *Trudy ISP RAN = Proceedings of ISP RAS*. 2021;33(3):87–100. (In Russ.). DOI: 10.15514/ISPRAS-2021-33(3)-7.
8. Lazarev V.A., Varlamov M.I., Yatskov A.K. Sistema upravleniya zadaniyami avtomatizirovannogo sbora dannykh iz seti Internet [Task management system for automated data collection from the Internet]. *Trudy ISP RAN = Proceedings of ISP RAS*. 2022;34(2):111–122. (In Russ.). DOI: 10.15514/ISPRAS-2022-34(2)-9.
9. Levashkin S.P., Ivanov K.N., Kushukov S.V. Ferma dannykh: informatsionnaya sistema sbora, khraneniya i obrabotki nestrukturirovannykh dannykh iz raznorodnykh istochnikov [Data farm: an information system for collecting, storing and processing unstructured data from heterogeneous sources]. *Trudy ISP RAN = Proceedings of ISP RAS*. 2023;35(2):57–72. (In Russ.). DOI: 10.15514/ISPRAS-2023-35(2)-5.
10. Serenko D.S., Terent'ev E.D., Zubarev D.V., Sochenkov I.V. Arkhitektura sistemy sbora i izvlecheniya informatsii dlya intellektual'noi poiskovo-analiticheskoi sistemy [Architecture of a data collection and extraction system for an intelligent search and analytical system]. *Trudy ISP RAN = Proceedings of ISP RAS*. 2025;37(2):263–280. (In Russ.). DOI: 10.15514/ISPRAS-2025-37(2)-20.
11. Lapkina A.V., Petukhov A.A. Klassifikatsiya HTTP-zaprosov k serveru v zadache avtomaticheskogo obkhoda sovremennykh veb-prilozhenii [Classification of HTTP requests to a server for automated crawling of modern web applications]. *Trudy ISP RAN = Proceedings of ISP RAS*. 2021;33(3):77–86. (In Russ.). DOI: 10.15514/ISPRAS-2021-33(3)-6.
12. Tacuri A., Firmenich S., Fernández A., Riva F., Urbietta M., Rossi G. Web Scraping by End Users. *IEEE Access*. 2025;13:205027–205044. DOI: 10.1109/ACCESS.2025.3636662.
13. Waseem M., Liang P., Shahin M. A Systematic Mapping Study on Microservices Architecture in DevOps. *Journal of Systems and Software*. 2020;170:110798. DOI: 10.1016/j.jss.2020.110798.
14. Zhou X., Li S., Cao L., Zhang H., Jia Z., Zhong C., et al. Revisiting the practices and pains of microservice architecture in reality: An industrial inquiry. *Journal of Systems and Software*. 2023;195:111521. DOI: 10.1016/j.jss.2022.111521.
15. Petushkov G.V., Sigov A.S. Analiz i vybor struktury mnogoprotsessornoi vychislitel'noi sistemy po kriteriyu bystrodeistviya [Analysis and selection of a multiprocessor computing system structure by the performance criterion]. *Russian Technological Journal*. 2024;12(6):20–25. (In Russ.). DOI: 10.32362/2500-316X-2024-12-6-20-25.
16. Giamattei L., Guerriero A., Pietrantuono R., Russo S., Malavolta I., Islam T., et al. Monitoring tools for DevOps and microservices: A systematic grey literature review. *Journal of Systems and Software*. 2024;208:111906. DOI: 10.1016/j.jss.2023.111906.
17. Alexandrescu A., Butincu C.N. Decentralized News-Retrieval Architecture Using Blockchain Technology. *Mathematics*. 2023;11(21):4542. DOI: 10.3390/math11214542.
18. Pan B., Stakhanova N., Ray S. Data Provenance in Security and Privacy. *ACM Computing Surveys*. 2023;55(14s):323. DOI: 10.1145/3593294.

19. Soriano-Salvador E., Guardiola-Múzquiz G. SealFS: Storage-based tamper-evident logging. *Computers & Security*. 2021;108:102325. DOI: 10.1016/j.cose.2021.102325.
20. Guardiola-Múzquiz G., Soriano-Salvador E. SealFSv2: combining storage-based and ratcheting for tamper-evident logging. *International Journal of Information Security*. 2023;22:447–466. DOI: 10.1007/s10207-022-00643-1.
21. Laaber C., Scheuner J., Leitner P. Software microbenchmarking in the cloud. How bad is it really? *Empirical Software Engineering*. 2019;24:2469–2508. DOI: 10.1007/s10664-019-09681-1.
22. Scrapy-Redis. Redis-based components for Scrapy. Documentation. URL: <https://scrapy-redis.readthedocs.io/en/master/readme.html> (accessed: 27.09.2026).
23. IST Research. Scrapy Cluster: distributed crawling architecture and implementation. URL: <https://github.com/istresearch/scrapy-cluster> (accessed: 27.09.2026).
24. ISO 28500:2017. Information and documentation — WARC file format. Geneva: International Organization for Standardization; 2017. URL: <https://www.iso.org/standard/68004.html> (accessed: 27.09.2026).
25. Laurie B., Messeri E., Stradling R. Certificate Transparency Version 2.0. RFC 9162. 2021. DOI: 10.17487/RFC9162.
26. Braun T.D., Siegel H.J., Beck N., Bölöni L.L., Maheswaran M., Reuther A.I., Robertson J.P., Theys M.D., Yao B., Hensgen D., Freund R.F. A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*. 2001;61(6):810–837. DOI: 10.1006/jpdc.2000.1714.
27. Cho J., Garcia-Molina H. Effective page refresh policies for Web crawlers. *ACM Transactions on Database Systems*. 2003;28(4):390–426. DOI: 10.1145/958942.958945.
28. Richards M., Ford N. *Fundamentals of Software Architecture: An Engineering Approach*. Sebastopol: O'Reilly Media; 2020. 432 p. ISBN: 9781492043447.
29. Koster M., Illyes G., Zeller H., Sassman L. Robots Exclusion Protocol. RFC 9309. 2022. DOI: 10.17487/RFC9309.
30. PostgreSQL Global Development Group. PostgreSQL 17 Documentation. Date/Time Functions and Operators. URL: <https://www.postgresql.org/docs/17/functions-datetime.html> (accessed: 27.09.2026).
31. YAML Language Development Team. *YAML Ain't Markup Language (YAML™)*, version 1.2.2. 2021. URL: <https://yaml.org/spec/1.2.2/> (accessed: 27.09.2026).