*Студенческий журнал по математике и её приложениям, 2025, Том 4, № 1. С. 26–32.*

# Эффективные подходы к проектированию архитектуры больших данных: ключевые принципы, современные технологии и примеры внедрения

**Крюкова А. Д.**
1472578@bsuedu.ru

**Аннотация.** Статья посвящена исследованию эффективных подходов к проектированию архитектуры больших данных, которые играют ключевую роль в обеспечении надежного хранения, обработки и анализа больших объемов информации. В работе рассматриваются основные принципы проектирования, включая масштабируемость, отказоустойчивость и гибкость архитектуры. Также проведен обзор современных технологий, таких как Apache Hadoop, Apache Kafka, Apache Spark и облачные решения (AWS, Google Cloud, Microsoft Azure), которые являются основой для реализации архитектур больших данных. Особое внимание уделено практическим аспектам внедрения архитектуры на основе реальных кейсов, включая системы аналитики в реальном времени, обработку потоковых данных и создание платформ для машинного обучения. Приведены рекомендации по выбору технологий и стратегий в зависимости от задач и объема данных.

**Ключевые слова:** архитектура больших данных, масштабируемость, обработка данных, Apache Hadoop, Apache Kafka, Apache Spark, потоковая аналитика, ETL-процесс, хранилище данных, облачные технологии, Лямбда-архитектура

**Для цитирования:** Крюкова А. Д. 2025. Эффективные подходы к проектированию архитектуры больших данных: ключевые принципы, современные технологии и примеры внедрения. *Студенческий журнал по математике и её приложениям*, 4(1): 26–32.

---

**1. Введение.** В современном мире объемы данных непрерывно увеличиваются, что делает управление ими одной из ключевых задач для бизнеса, науки и технологий. Термин «большие данные» становится основой для принятия стратегических решений, построения интеллектуальных систем и оптимизации бизнес-процессов. Однако работа с большими данными требует не только мощных технологий, но и грамотного проектирования архитектуры, способной обеспечить надежность, масштабируемость и эффективность обработки данных.

Архитектура больших данных представляет собой совокупность подходов, принципов и технологий, которые позволяют собирать, обрабатывать, хранить и анализировать огромные объемы информации. При ее проектировании важно учитывать не только технические ограничения, но и потребности конкретного бизнеса, а также современные тенденции и инструменты. Например, распределенные системы хранения, такие как Hadoop HDFS, и платформы для обработки потоковых данных, такие как Apache Kafka, стали неотъемлемой частью современных решений.

Цель данной статьи — предложить обзор эффективных подходов к проектированию архитектуры больших данных, рассмотреть ключевые принципы, такие как отказоустойчивость, гибкость и производительность, а также проанализировать современные технологии и успешные примеры их внедрения. Особое внимание будет уделено практическим рекомендациям по выбору инструментов и реализации архитектуры.

В статье предоставляется подробное описание всех этапов построения архитектуры больших данных, начиная с выбора технологий и заканчивая их интеграцией в реальную бизнес-среду.

**2. Основные сведения.** «Проектирование архитектуры больших данных — это процесс создания структурированной системы, способной эффективно принимать, обрабатывать и анализировать данные, которые являются слишком объемными или сложными для традиционных систем баз данных» [3, 10]. Этот процесс включает в себя определение компонентов системы, таких как источники данных, хранилища, механизмы обработки и аналитические инструменты, а также установление взаимодействия между ними (рис. 1).

Ключевыми аспектами проектирования архитектуры больших данных являются:

1. Масштабируемость – обеспечение способности системы обрабатывать увеличивающиеся объемы данных без снижения производительности.

2. Отказоустойчивость – построение системы, способной продолжать функционировать при сбоях отдельных компонентов.

3. Гибкость – возможность адаптации системы к изменяющимся требованиям и интеграции новых технологий.

4. Производительность – оптимизация скорости обработки и анализа данных для своевременного получения результатов [2].

© Крюкова А. Д., 2025

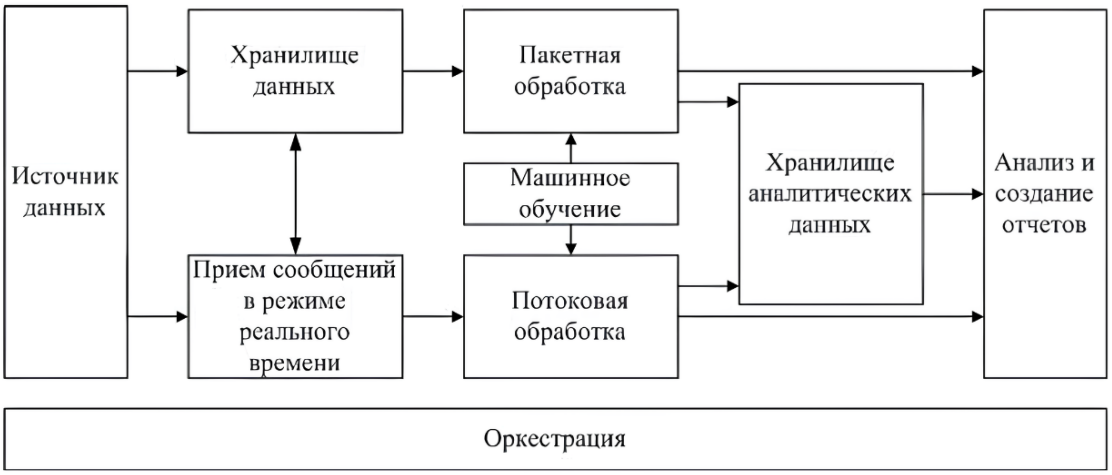


Рис 1. Компоненты архитектуры для обработки больших данных

Рассмотрим диаграмму распределения типов данных в структуре больших данных (рис. 2).

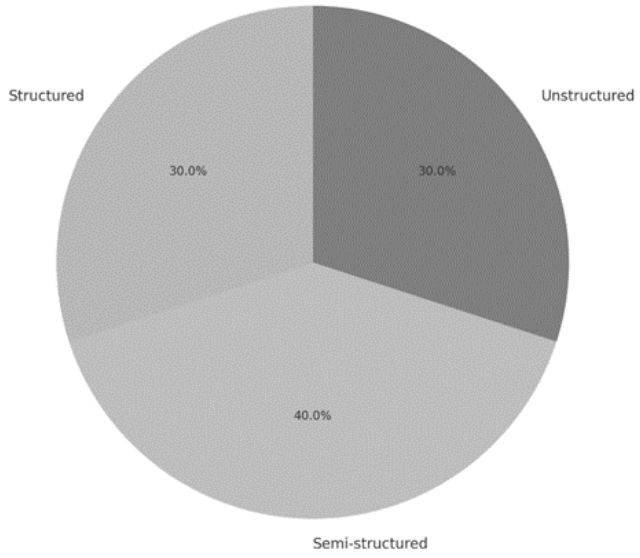


Рис 2. Компоненты архитектуры для обработки больших данных

Structured (Структурированные данные): 30%. Это данные, организованные в таблицах, например, SQL-базы данных:

1. Транзакции в реляционных базах данных (MySQL, PostgreSQL, Oracle).
2. Финансовые отчёты и инвентарные системы.

Согласно отчету IDC, структурированные данные составляют около 30% от общего объема данных в корпоративной среде, так как такие данные часто хранятся в традиционных базах данных [13].

Semi-structured (Полуструктурированные данные): 40%. Данные с определенной структурой, такие как JSON, XML:

1. Логи веб-серверов (например, формат JSON).
2. Файлы конфигурации (YAML, XML).
3. Данные IoT-устройств.

Исследование Gartner указывает, что данные из систем IoT, социальных сетей и веб-приложений всё чаще поступают в полуструктурированном виде, занимая более 40% от общего объема данных в больших системах [14].

Unstructured (Неструктурированные данные): 30%. Данные без четкой схемы, например, изображения, видео, аудио:

1. Видео, изображения (медицинские снимки).

2. Аудио (подкасты, голосовые помощники).

3. Тексты (электронная почта, форумы).

Анализ компании Seagate показал, что более 30% данных, генерируемых пользователями, приходится на медиафайлы, включая видео и изображения [17].

Проектирование архитектуры больших данных требует применения эффективных подходов, обеспечивающих надежность, масштабируемость и производительность системы.

К ключевым подходам можно отнести:

1. Выбор технологий – использование современных инструментов, таких как Apache Hadoop, Apache Spark и Apache Kafka, позволяет эффективно обрабатывать и анализировать большие объемы данных.

2. Модульность и гибкость – построение системы из независимых компонентов облегчает ее модификацию и масштабирование в ответ на изменяющиеся бизнес-требования.

3. Обеспечение безопасности и качества данных – внедрение строгих мер безопасности и процедур обеспечения качества данных гарантирует их целостность и защищенность.

4. Интеграция потоковой и пакетной обработки – комбинирование этих методов позволяет обрабатывать данные в реальном времени и анализировать их в пакетном режиме для получения более глубоких инсайтов.

5. Использование облачных решений – облачные платформы предоставляют гибкость и ресурсы для хранения и обработки больших данных, обеспечивая при этом высокую доступность и отказоустойчивость [5].

Применение этих подходов способствует созданию эффективной архитектуры больших данных, способной удовлетворить современные требования к обработке и анализу информации.

«Современные технологии обработки больших данных предоставляют мощные инструменты для сбора, хранения и анализа информации. Рассмотрим некоторые из них:

Apache Hadoop — это фреймворк с открытым исходным кодом, предназначенный для распределенного хранения и обработки больших объемов данных. Его основными компонентами являются Hadoop Distributed File System (HDFS) для хранения данных и MapReduce для их обработки. Hadoop обеспечивает горизонтальное масштабирование и отказоустойчивость, что делает его популярным выбором для построения хранилищ данных и аналитических платформ » [7, 15].

«Apache Kafka — это распределенная платформа потоковой передачи данных, разработанная для обработки и передачи большого количества сообщений в реальном времени. Kafka используется для создания надежных и масштабируемых конвейеров данных, обеспечивая высокую пропускную способность и низкую задержку. Она широко применяется для сбора логов, мониторинга, а также в системах, требующих обработки событий в реальном времени» [7, 16].

«Apache Spark — это унифицированный аналитический движок с открытым исходным кодом для обработки больших данных, известный своей скоростью и простотой использования. Spark поддерживает различные модели вычислений, включая пакетную обработку, интерактивные запросы, обработку потоков данных и машинное обучение. Его способность к быстрой обработке данных делает его предпочтительным выбором для аналитических задач, требующих высокой производительности» [7, 15].

«Облачные платформы (AWS, Google Cloud, Microsoft Azure) предоставляют широкий спектр услуг для хранения, обработки и анализа данных, обеспечивая гибкость и масштабируемость без необходимости в управлении физической инфраструктурой.

Amazon Web Services (AWS): Одна из первых и наиболее зрелых облачных платформ, предлагающая более 200 полнофункциональных сервисов, включая вычислительные ресурсы, хранение данных и аналитические инструменты [8].

Google Cloud Platform (GCP): Облачная платформа от Google, известная своими передовыми возможностями в области аналитики и машинного обучения, а также интеграцией с другими сервисами Google.

Microsoft Azure: Облачная платформа от Microsoft, предлагающая широкий спектр услуг и тесную интеграцию с продуктами Microsoft, что делает ее привлекательной для предприятий, использующих экосистему Microsoft» [12].

Выбор между этими технологиями и платформами зависит от конкретных требований проекта, объемов данных, бюджета и существующей инфраструктуры. Комбинированное использование этих

инструментов позволяет создавать мощные и гибкие решения для обработки и анализа больших данных.

Сравним два брокера Apache Kafka и RabbitMQ.

«RabbitMQ Написан на Erlang и совместим с большинством популярных ОС. Брокер берет на себя много дополнительных обязательств. Например, следит за прочитанными сообщениями и удаляет их из очереди. Или сам организует процесс распределения сообщений между подписчиками.

Apache Kafka позиционирует себя не столько брокером сообщений, сколько стриминговой платформой. То есть, кроме распределений сообщений Kafka еще может использоваться для поиска событий, отслеживания веб-активности и метрик. Также ее можно использовать как базу данных. Сама Kafka написана на Java и Scala компанией LinkedIn. Так, в отличие от RabbitMQ, Kafka не занимается контролем и распределением сообщений, то есть, не выполняется лишняя логика в брокере. Благодаря этому Kafka специализируется на высокой пропускной способности данных и низкой задержке для обработки потоков данных в реальном времени» [16].

Процентная нагрузка по пропускной способности определяется по формуле:

$$\text{Load percentage} = \frac{\text{Throughput}}{\text{Maximum Capacity}} \times 100, \tag{1}$$

где: Throughput – фактическое количество обработанных сообщений. Maximum Capacity – теоретический предел системы.

В контексте современных архитектур данных, Лямбда и Каппа архитектуры выделяются своей спецификой. Они представляют собой подходы к построению ETL (Extract, Transform, Load) pipelines, что является ключевым элементом в обработке данных. Рассмотрим подробнее принципы работы лямбда-архитектуры.

Лямбда-архитектура обеспечивает обработку больших объемов данных с минимальной задержкой, выделяясь своей отказоустойчивостью и возможностью масштабирования. Это достигается за счет сочетания пакетной обработки и скоростного анализа, позволяя интегрировать новые данные, сохраняя при этом целостность и доступность исторической информации. Такие качества делают лямбда-архитектуру востребованной в реальных Big Data проектах [9, 11].

Можно выделить два основных компонента Лямбда-архитектуры: пакетную, так и потоковую обработку данных.

«Пакетная обработка данных – это процесс обработки данных в больших объемах, сгруппированных в небольшие партии (батчи). Этот подход идеально подходит для анализа больших объемов статических данных, таких как исторические записи или данные, которые обновляются нечасто [6].

В пакетной обработке данные загружаются, обрабатываются, и результаты сохраняются в батчах. Это позволяет эффективно обрабатывать большие объемы данных в ограниченное время» [1, 4].

Пример кода для пакетной обработки данных, использующего Hadoop FileSystem API для взаимодействия с файловой системой. Этот код демонстрирует чтение файлов из HDFS, выполнение обработки данных и сохранение результата обратно в HDFS:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IOUtils;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.OutputStream;

public class BatchProcessingExample {
    public static void main(String[] args) throws Exception {
        // 1. Настройка конфигурации Hadoop
        Configuration conf = new Configuration();
        conf.set("fs.defaultFS", "hdfs://localhost:9000"); // Укажите ваш
            HDFS URI

        // 2. Создание файловой системы
        FileSystem fs = FileSystem.get(conf);

        // 3. Пути к входным и выходным данным
        Path inputPath = new Path("/input/batch-data.txt");
        Path outputPath = new Path("/output/processed-data.txt");

        // 4. Чтение данных из HDFS
```

```
            if (!fs.exists(inputPath)) {
                System.out.println("Input file does not exist: " + inputPath);
                return;
            }

            BufferedReader reader = new BufferedReader(new InputStreamReader(fs.
                open(inputPath)));
            String line;
            StringBuilder processedData = new StringBuilder();

            // 5. Обработка данных
            while ((line = reader.readLine()) != null) {
                // Пример: преобразование текста в верхний регистр
                processedData.append(line.toUpperCase()).append("\n");
            }

            // 6. Запись обработанных данных обратно в HDFS
            if (fs.exists(outputPath)) {
                fs.delete(outputPath, true); // Удаляем, если файл уже
                    существует
            }

            OutputStream outputStream = fs.create(outputPath);
            outputStream.write(processedData.toString().getBytes());

            // 7. Закрытие ресурсов
            IOUtils.closeStream(reader);
            IOUtils.closeStream(outputStream);

            System.out.println("Batch processing completed. Processed data saved
                 to: " + outputPath);
        }
    }
```

Потоковая обработка данных позволяет анализировать данные в реальном времени по мере их поступления. Это особенно важно для систем, требующих низкой задержки между событием и его обработкой, таких как мониторинг и анализ логов.

Потоковая обработка данных включает в себя непрерывный прием данных, их обработку и анализ в реальном времени. Это позволяет мгновенно реагировать на события и принимать решения»[1, 4].

Пример потоковой обработки данных с использованием Apache Kafka для чтения сообщений из топика, обработки данных и сохранения результата в файловую систему.

```
from kafka import KafkaConsumer
import os

# Конфигурация Kafka
KAFKA_BROKER = "localhost:9092"
TOPIC = "streaming-topic"
OUTPUT_DIR = "output"

# Убедимся, что директория для сохранения данных существует
if not os.path.exists(OUTPUT_DIR):
os.makedirs(OUTPUT_DIR)

# Создание Kafka Consumer
consumer = KafkaConsumer(
TOPIC,
bootstrap_servers=[KAFKA_BROKER],
auto_offset_reset="earliest",  # Чтение сообщений с начала топика
enable_auto_commit=True,
group_id="stream-processing-group"
)

print(f"Listening to topic: {TOPIC}")

# Чтение сообщений и обработка
```

```
    with open(os.path.join(OUTPUT_DIR, "processed_data.txt"), "w") as
        output_file:
    for message in consumer:
    # Декодируем сообщение
    raw_data = message.value.decode("utf-8")

    # Пример обработки данных (преобразование в верхний регистр)
    processed_data = raw_data.upper()

    # Записываем результат в файл
    output_file.write(processed_data + "\\n")
    output_file.flush()  # Обеспечиваем мгновенную запись на диск

    print(f"Processed message: {processed_data}")
```

Таким образом, использовать пакетную обработку следует когда данные не требуют обработки в реальном времени. Обработка исторических данных для анализа и прогнозирования:

1. Аналитика данных продаж за месяц.
2. Подготовка данных для моделей машинного обучения.

Потоковая обработка применяется, когда важно реагировать на данные в реальном времени. Мониторинг событий и работа с потоками данных:

1. Фрод-детекция в банковской системе.
2. Рекомендации пользователям на основе текущей активности.

**6. Заключение.** Эффективное проектирование архитектуры больших данных требует не только глубокого понимания технологий, но и анализа конкретных задач. Современные научные исследования и практика показывают, что сочетание традиционных (пакетных) и инновационных (потоковых) подходов, а также использование облачных технологий открывает новые возможности для решения сложных задач анализа и обработки данных. Применение принципов масштабируемости, отказоустойчивости и гибкости позволяет построить системы, которые удовлетворяют как текущие, так и будущие потребности в управлении данными.

### Список литературы


1. Архитектурный паттерн для обработки больших данных: Lambda [Электронный ресурс]. – Режим доступа: https://habr.com/ru/companies/otus/articles/766672/ (дата обращения 07.01.2025).
2. Баранова С. Н. Проектирование архитектур систем работы с большими данными // Современные технологии в науке и образовании. – Рязань, 2021. – С. 154.
3. Варианты архитектуры для обработки больших данных [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/ru-ru/azure/architecture/databases/guide/big-data-architectures (дата обращения 07.01.2025).
4. Григорьев Ю. А., Ермаков О. Ю. Обработка запросов в системе с лямбда-архитектурой на уровне ускорения // Информатика и системы управления. – 2020. – № 2. – С. 3–16.
5. Клеменков П. А., Кузнецов С. Д. Большие данные: современные подходы к хранению и обработке // Труды ИСП РАН. – 2012. – 143–156 с.
6. Матвеева П. Р. Сравнение лямбда и традиционной архитектур // Форум молодых ученых. – 2018. – № 1(17). – С. 734–740.
7. Методы ситуационного анализа и графической визуализации потоков больших данных. – Пролетарский А. В., Березкин Д. В., Гапанюк Ю. Е. // Вестник МГТУ им. Н.Э. Баумана. Серия «Приборостроение». – М., 2018. – № 2(119). – 98–123 с.
8. Новиков Б. А., Графеева Н. Г., Михайлова Е. Г. Big data: Новые задачи и современные подходы // КИО. – 2014. – № 4. – С. 10–18.
9. Осипов Д. Технологии проектирования баз данных. – Litres, 2022.
10. Понин Ф. Н. Методология проектирования и создания баз данных для современного программного обеспечения // Universum: технические науки. 2024. №1 (118). URL: https://cyberleninka.ru/article/n/metodologiya-proektirovaniya-i-sozdaniya-baz-dannyh-dlya-sovremennogo-programmnogo-obespecheniya (дата обращения: 07.01.2025).

11. Сокольников А. М. Сравнительный анализ подходов к разработке архитектуры и систем управления базами данных для высоконагруженных WEB-сервисов // Кибернетика и программирование. – 2014. – №. 4. – С. 1–13.
12. AWS vs Azure vs Google Cloud [Электронный ресурс]. - Режим доступа: https://cloudfresh.com/ru/cloud-blog/aws-vs-azure-vs-google-cloud/ (дата обращения 07.01.2025).
13. Data Age 2025 [Электронный ресурс]. – Режим доступа: https://www.seagate.com/www-content/our-story/trends/files/Seagate-WP-DataAge2025-March-2017.pdf (дата обращения 07.01.2025).
14. Gartner [Электронный ресурс]. – Режим доступа: https://www.gartner.com/en (дата обращения 07.01.2025).
15. Hadoop vs. Spark: What's the difference? [Электронный ресурс]. – Режим доступа: https://www.ibm.com/think/insights/hadoop-vs-spark (дата обращения 07.01.2025).
16. Kafka vs. Spark vs. Hadoop [Электронный ресурс]. – Режим доступа: https://www.logicmonitor.com/blog/kafka-vs-spark-vs-hadoop (дата обращения 07.01.2025).
17. Seagate Reports [Электронный ресурс]. – Режим доступа: https://www.seagate.com/ (дата обращения 07.01.2025).

*Поступила в редакцию 28.01.2025*

---

**СВЕДЕНИЯ ОБ АВТОРЕ**

**Крюкова Анжелика Дмитриевна** – магистрант 1-го года обучения, Белгородский государственный национальный исследовательский университет

**РУКОВОДИТЕЛЬ**

**Чернова Ольга Викторовна** – кандидат физико-математических наук, доцент, доцент кафедры прикладной математики и компьютерного моделирования, Белгородский государственный национальный исследовательский университет

Chernova_Olga@bsuedu.ru

К содержанию