

МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования

«ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ»

Факультет

вычислительной техники

Кафедра

«Вычислительная техника»

Направление подготовки: 09.04.01 «Информатика и вычислительная техника»

Магистерская программа: «Прикладной искусственный интеллект»

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

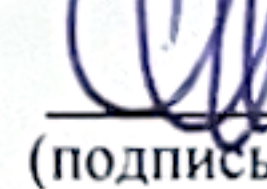
на тему

**ОПТИМИЗАЦИЯ АЛГОРИТМОВ ДЛЯ
ВЫЧИСЛЕНИЯ ХЕШ-КОДА В ЯЗЫКЕ JAVA**

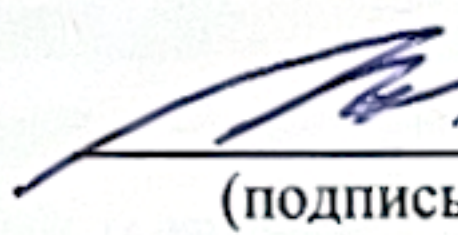
Студент

 23.06.2025 Ахунов Марат Алмазович
(подпись, дата) (ФИО полностью)

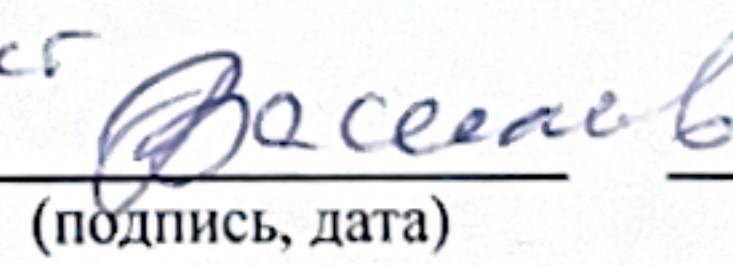
Руководитель

 23.06.2025 Какзков Б.В.
(подпись, дата) (фамилия, инициалы)

Нормоконтролёр

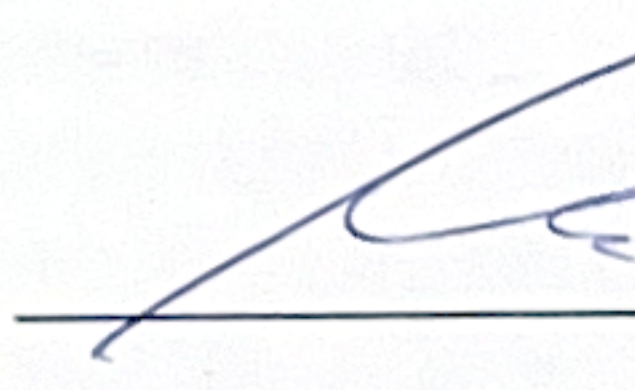
 23.06.2025 Бычков А.С.
(подпись, дата) (фамилия, инициалы)

Рецензент

главный специалист
АО «ИПТ Рудник»  23.06.2025 Вассильев А.В.
(должность, место работы) (подпись, дата) (фамилия, инициалы)

Работа допущена к защите (протокол зас. кафедры от 28.05.2025 № 16)

Заведующий кафедрой

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

Работа защищена с отметкой хорошо (протокол заседания ГЭК от

25 июня 2025 г. № 12)

Секретарь ГЭК

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

Пенза 2025

УТВЕРЖДАЮ

Зав. кафедрой ВТ

М.А. Митрохин

«29» 09 2023 г.

ЗАДАНИЕ

на выполнение магистерской работы

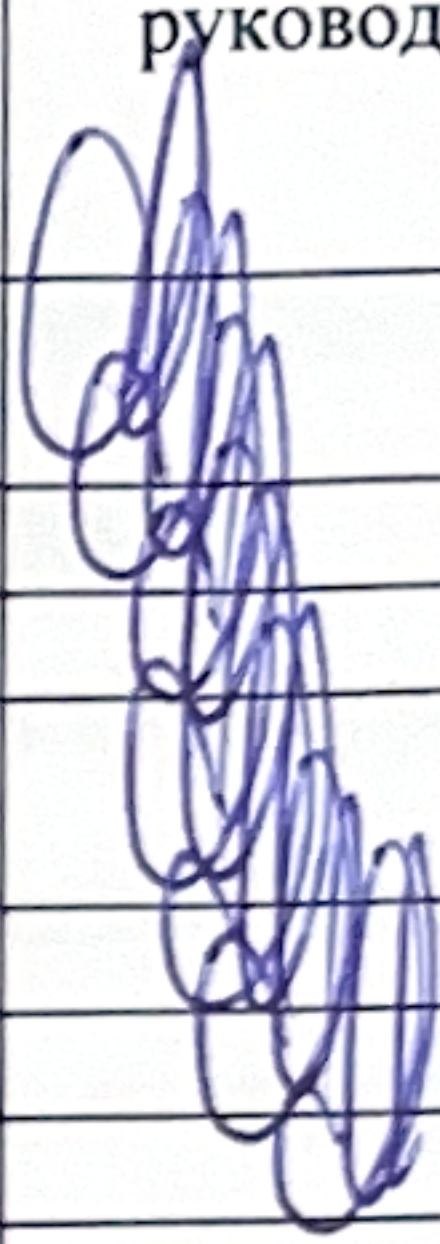
1. Студент гр. 23ВВИМ1 факультета ВТ направление 09.04.01
Ахунов Марат Алмазович
(фамилия, имя, отчество)
2. Руководитель ВКР доцент кафедры МОиПЭВМ, ПГУ
(Должность и место работы)
Казаков Борис Владимирович
(фамилия, имя, отчество)
3. Время выполнения магистерской работы с _____
по _____
4. Тема магистерской работы Оптимизация алгоритмов для вычисления хеш-кода в языке Java
Тема утверждена приказом ПГУ № 986/0 от 29.09.2023 г.
5. Задание на магистерскую работу: предмет и назначение разработки, функциональные, программные, технические требования и т.п.
 - 5.1. Разрабатываемое программное обеспечение предназначено для интеллектуального выбора оптимальной хеш-функции и генерации хеш-кода с использованием нейросетевой модели, с целью повышения эффективности и надёжности хеширования.
 - 5.2 Требования к функциональности: Программное обеспечение позволяет:
осуществлять загрузку JSON-объектов через веб-интерфейс;
осуществлять классификацию входных данных и выбор оптимальной хеш-функции на основе модели машинного обучения;
осуществлять генерацию хеш-кода при помощи нейросетевой модели;
визуализировать результат вычислений;
проводить сравнительный анализ полученного решения и других функций;
 - 5.3 Требования к ИО: должен быть использовать формат данных JSON, использующийся для генерации хеш-кода и рекомендации алгоритма.
 - 5.4 Требования к ПО: IntelliJ IDEA, PyCharm ONNX Runtime, Java Python, PHP, PostgreSQL, JSON, Java Script
 - 5.5 Требования к аппаратному обеспечению: Операционная система Microsoft Windows 10 или Ubuntu 20.04, процессор Intel Core i5-7400 или мощнее.
6. Содержание пояснительной записки магистерской работы: перечень подлежащих разработке, расчетов, обоснований, описаний
 - 6.1 Анализ алгоритмов для вычисления хеш-кода
 - 6.2 Выбор средств разработки

6.3 Структура системы

6.4 Разработка программного обеспечения

6.5 Тестирование

7. Календарный график выполнения магистерской работы

Наименование этапов работы	Объем работы	Срок выполнения	Подпись руководителя
1. Анализ алгоритмов для вычисления хеш-кода	15 %	24.11.2023	
2. Выбор средств разработки	10 %	15.01.2024	
3. Структура системы	15 %	11.03.2024	
4. Разработка программного обеспечения	35%	26.06.2024	
5. Тестирование	15 %	15.03.2025	
6. Оформление ПЗ	10 %	15.06.2025	
7. Норм контроль		23.06.2025	

Дата выдачи задания

«29» 09 2023

Руководитель магистерской работы

Казаков Б.В.

(фамилия, имя, отчество)

Задание к исполнению принял

«29» 09 2023

Студент

Ахунов М.А.

(фамилия, имя, отчество)

Магистерскую работу к защите допустить

Декан ФВТ

 24.06.2025

/ Л.Р. Фионова

(Подпись, дата)

РЕФЕРАТ

Пояснительная записка содержит 105 листов, 32 рисунка, 2 таблицы, 34 источника, 3 приложения.

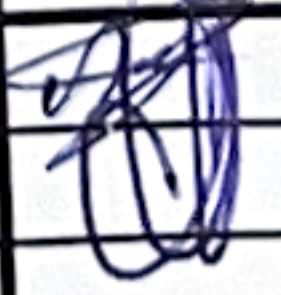
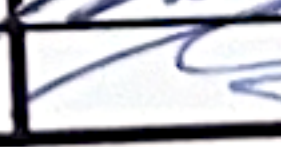
ОПТИМИЗАЦИЯ ХЕШИРОВАНИЯ, НЕЙРОННЫЕ СЕТИ, ХЕШ-ФУНКЦИЯ, НЕЙРОСЕТЕВАЯ МОДЕЛЬ, JSON, JAVA, PYTHON, SPRING BOOT, REACT, POSTGRESQL, FLASK, HASHING

Объектом разработки является веб-приложение, предназначенное для интеллектуального выбора и генерации оптимального хеш-кода.

Цель работы – оптимизация алгоритмов хеширования структурированных данных с использованием методов искусственного интеллекта, в том числе автоматизация выбора наиболее эффективной хеш-функции и генерация хеш-кодов с повышенной уникальностью, стабильностью и равномерностью.

При разработке были последовательно решены следующие задачи: изучены криптографические и некриптографические хеш-функции, проведён анализ существующих алгоритмов и средств хеширования, выбран и обоснованы используемые технологический, разработана архитектура приложения и реализованы программные модули, включая нейросетевой классификатор и генератор хешей.

В результате выполненной работы создано распределённое веб-приложение, включающее серверную часть на Java, клиентский интерфейс и модуль ИИ.

					ПГУ 1.09.04.01.03.001 ПЗ				
Изм.	Лист	№ докум.	Подпись	Дата	Оптимизация алгоритмов для вычисления хеш-кода в языке Java Пояснительная записка	Лит.	Лист	Листов	
Разраб.		Ахунов М.А.		23.06.25					
Провер.		Казаков Б.В.		23.06.25			2	105	
Н. Контр.		Бычков А.С.		23.06.25		гр. 23ВВИм1			
Утв.		Митрохин М.А.		23.06.25					

СОДЕРЖАНИЕ

_Тос201618954ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ	5
ВВЕДЕНИЕ	6
1 Анализ алгоритмов для вычисления хеш-кода	9
1.1 Криптографические хеш-функции	9
1.1.1 Криптографическая функция SHA-256.....	11
1.1.2 Криптографический алгоритм ГОСТ Р 34.11-94	14
1.2 Некриптографические хеш-функции	19
1.2.1 Базовая хеш-функция в Java.....	23
1.2.2 Хеш-функция FNV-1a.....	26
1.2.3 Хеш-функция MurmurHash3	29
1.2.4 Хеш-функция CRC32	33
2 Выбор средств разработки	41
2.1 Выбор технологий для backend части	41
2.2 Модуль для искусственного интеллекта.....	42
2.3 Библиотеки хеширования в Java.....	44
2.4 Выбор технологий для визуальной части приложения.....	46
2.5 Выбор базы данных.....	48
2.6 Выбор инструментов для тестирования и разработки	51
2.7 Средства развертывания приложения.....	54
3 Структура системы.....	56
3.1 Основные функции подсистемы.....	56
3.2 Архитектура и компоненты системы	59
4 Разработка программного обеспечения	63
4.1 Разработка веб сервиса	63
4.2 Разработка клиентской части.....	67
4.3 Разработка сервиса искусственного интеллекта.....	68
5 Тестирование	74
Заключение	81
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	83

ПРИЛОЖЕНИЕ А Листинг серверной части приложения.....	86
ПРИЛОЖЕНИЕ Б Листинг модуля искусственного интеллекта.....	95
ПРИЛОЖЕНИЕ В Презентация	102

ПЕРЕЧЕНЬ СОКРАЩЕНИЙ И ОБОЗНАЧЕНИЙ

ГОСТ – Государственный стандарт

ИИ – искусственный интеллект

GUID (Globally Unique Identifier) – Глобально уникальный идентификатор

IDE (Integrated Development Environment) – Интегрированная среда разработки

JSON (JavaScript Object Notation) – Формат обмена данными

ML (Machine Learning) – Машинное обучение

VCS (Version Control System) – Система контроля версий

ВВЕДЕНИЕ

Хеширование данных является одной из важных направлений современных алгоритмов и структур данных. Хеширование применяется повсеместно, начиная от применения в простых структурах данных, заканчивая применением при работе с большим количеством информации для ускорения процессов вычисления и сокращению времени работы программ [1]. Хеширование – это метод преобразования данных любого размера в строку фиксированной длины с помощью специальной математической функции – хеш-функции. Под хеш-функцией обычно понимают алгоритм преобразования входных данных в фиксированное значение – хеш-код.

Хеш-код используется для быстрого поиска или сравнения данных. Например, в структурах данных на языке Java он определяет позицию объекта в массиве данных, а также по нему производят сравнение между одинаковыми объектами [2]. Ключевое свойство хорошей хеш-функции – это равномерное распределение входных значений, сводя к минимуму количество коллизий. Случай, когда разные входные данные получают одинаковый хеш-код, называется коллизией, такие случаи не желательны, так как они снижают эффективность поиска, ведь вместо прямого доступа к объекту приходится обрабатывать несколько элементов, что замедляет работу программы.

Однако создать идеальную и универсальную хэш-функцию непросто. Традиционные хэш-функции часто используют простые математические операции и стремятся вычислить псевдослучайное значение по отношению к входным данным. Это означает, что без дополнительной информации о природе данных они могут генерировать коллизии практически случайным образом. Существуют совершенные хэш-функции – это специальные конструкции, которые вообще не дают коллизий для фиксированного набора

данных, но для корректной работы они требуют предварительного знания набора и обычно тяжелы в вычислительном отношении [3].

На практике разработчики используют разнообразные компромиссные решения, например, быстрые не криптографические хэш-функции, достаточно хорошо распределяющие значения для среднего случая, и более сложные криптографические хэш-функции, генерирующие неповторяющиеся хеш-коды, но тратя на их создание на порядок больше времени.

В последние годы появилось много исследований, посвящённых использованию методов искусственного интеллекта для оптимизации алгоритмов и структур данных. Например, группа исследователей из MIT показала, что в некоторых случаях замена традиционного хеширования на модель, обученную на конкретном наборе данных, позволяет сократить число коллизий почти вдвое.

Основной целью работы является исследование и применение методов искусственного интеллекта для оптимизации алгоритмов вычисления хеш-кода в языке программирования Java. В первую очередь снижения количества коллизий и повышения равномерности распределения значений без ущерба для скорости вычисления хеш-кода.

Основными задачи работы является:

- изучение существующих хеш-функций в java,
- изучить существующие подходы к оптимизации хэш-функций с помощью методов искусственного интеллекта,
- рассмотреть возможные архитектуры и инструменты реализации решений на основе ИИ в связке с Java,
- разработать решение для вычисления хеш-кода на основе искусственного интеллекта,
- апробация реализованного функционала в форме тестирования.

Результаты работы могут представлять интерес для разработчиков систем хранения и обработки данных, где особенно важна эффективность хеширования. Предложенный подход также открывает возможности для

дальнейших исследований в области применения методов искусственного интеллекта при построении хеш-функций.

1 Анализ алгоритмов для вычисления хеш-кода

Хэш-функция – это функция, также называемая функцией свёртки, преобразующая массив входных данных произвольного размера в битовую строку фиксированной длины по определённому алгоритму. Хеш-функцию можно представить, как отображение, представленное на рисунке 1.

$$H : \{0, 1\}^* \rightarrow \{0, 1\}^n$$

Рисунок 1 – Отображение хеш-функции

Левая часть отображения является множеством битовых строк произвольной длины, а правая множество битовых строк фиксированной длины или выходной хеш-код. Процесс вычисления называется хешированием, а результат хешем исходных данных. Благодаря сжатию произвольных данных до фиксированной длины хеширование используется для многих задач: от ускорения поиска данных до проверки целостности файлов. Качество хэш-функции во многом определяется малой вероятностью случайной коллизии, а также способностью затруднять преднамеренный подбор коллизующих входов [4]. Различают два основных класса хэш-функций – криптографические и не криптографические. Ниже рассматривается, чем они отличаются, каковы их цели, ключевые свойства и требования.

1.1 Криптографические хеш-функции

Криптографические хеш-функции – это разновидность хеш-функций, предназначенная для задач, связанных с защитой информации. Их основным назначением является обеспечение целостности и подтверждение подлинности данных. Даже незначительное изменение исходного сообщения приводит к заметному изменению хеш-кода, что позволяет обнаружить факт подмены. Такие функции находят широкое применение при создании цифровых подписей, проверке целостности файлов, хранении паролей и в

других областях, где важна безопасность данных [5]. Например, в системе электронной подписи подписывается именно хеш документа, и стойкость хеш-функции к коллизиям необходима, чтобы злоумышленник не смог подделать другой документ с тем же хеш-кодом.

Чтобы использоваться в целях информационной безопасности, криптографическая хеш-функция должна обладать рядом устойчивых характеристик:

- необратимость,
- устойчивость к коллизиям,
- лавинный эффект,
- псевдослучайность.

Данные свойства являются основой для создания хеш-функций. Необратимость подразумевает под собой то, что по известному значению хеш-кода невозможно восстановить или подобрать исходные данные. Это свойство делает хеш-функцию односторонней. Как было подмечено ранее при хешировании количество коллизий должно стремиться к нулю, криптографическая функция считается стойкой, если результат невозможно подобрать за разумное время, функция должна быть устойчива к коллизиям. Также хеш-функция должна иметь свойство лавинного эффекта, под ней подразумевается то, что при малейшем изменении входных данных приводит к существенному и непредсказуемому изменению выходного хеш-кода [5].

Перечисленные свойства формализованы в современных стандартах. Например, требованиям к криптографическим хэш-функциям ГОСТ Р 34.11-2012 (алгоритм «Стрибог») включают стойкость к вычислению прообраза, второго прообраза, коллизий, а также требование, чтобы функция не была дополняема, то есть невозможно было предсказать хеш составного сообщения по хешам частей. В практическом плане криптографические хэши обычно имеют достаточно большой размер вывода [6]. Чем больше бит содержит хеш, тем сложнее подбор коллизий перебором. Поэтому стандартные криптографические алгоритмы (SHA-256, SHA-3, BLAKE2 и

др.) выдают хеш длиной 256 бит или более – этого достаточно, чтобы подбор коллизий был практически невозможен [7].

Также одним из важнейших требований является устойчивость ко всем известным видам криптоаналитических атак. Хэш-функция не должна иметь уязвимостей, которые снижали бы сложность поиска коллизий или прообразов по сравнению с грубым перебором. В истории некоторые криптографические хэш-функции были взломаны, был найден метод построения коллизий значительно быстрее простого перебора. Это был не единичный случай, такие функции являются недостаточно стойкими и не рекомендуются для применения в защите данных. Современные криптографические функции на сегодняшний день считаются вычислительно стойкими. На настоящий момент каких-либо коллизий или эффективных атак не было обнаружено. Подбор прообразов для таких функций с имеющимися на текущий момент ресурсами считается невозможным. Далее будут представлены примеры функций.

1.1.1 Криптографическая функция SHA-256

SHA-256 – одна из самых известных криптографических хэш-функций, принадлежащая семейству SHA-2 (Secure Hash Algorithm 2). Она была разработана Агентством национальной безопасности США (NSA) и стандартизована NIST в 2001 году. SHA-256 на вход принимает сообщение любой длины (меньше 2^{64} бит) и вычисляет 256-битный (32 байта) хеш-код. Основой SHA-256 является классическая конструкция Меркла-Дамгора (Merkle–Damgård): вход разбивается на блоки фиксированного размера (512 бит), которые последовательно обрабатываются компрессионной функцией, обновляющей внутреннее состояние. Внутреннее состояние SHA-256 – это 256-бит (8 слов по 32 бита), которые инициализируются определёнными константами. Каждый 512-битный блок сообщения проходит через 64 раунда преобразований, в каждом из которых используются нелинейные логические функции (битовые операции AND, XOR, NOT) и арифметические складки по модулю 2^{32} с добавлением постоянных круглых констант.

Первый шаг в алгоритме хеширования дописывает бит, в качестве значения использует единицу, к исходным данным, затем столько нулей, чтобы длина стала конгруэнтной, то есть должно не хватать 64 бита для кратности 512. В последние 64 бита дописывается исходная длина сообщения, в результате первого шага получается строка, кратная 512 битам. Вторым шагом происходит инициализация внутренних восьми 32 битных слов, то есть регистры A, B, C, D, E, F, G, H устанавливаются в заранее определенные константы. Константами в данном контексте являются фрагменты дробных частей квадратных корней из простых чисел – исторически зафиксированные значения. Третий шаг алгоритма для каждого 512-битного блока сообщения выполняет 64 шага обновления регистров. На рисунке 2 показана блок-схема одного раунда компрессионной функции SHA-256, иллюстрирующая операции над 32-битными словами.

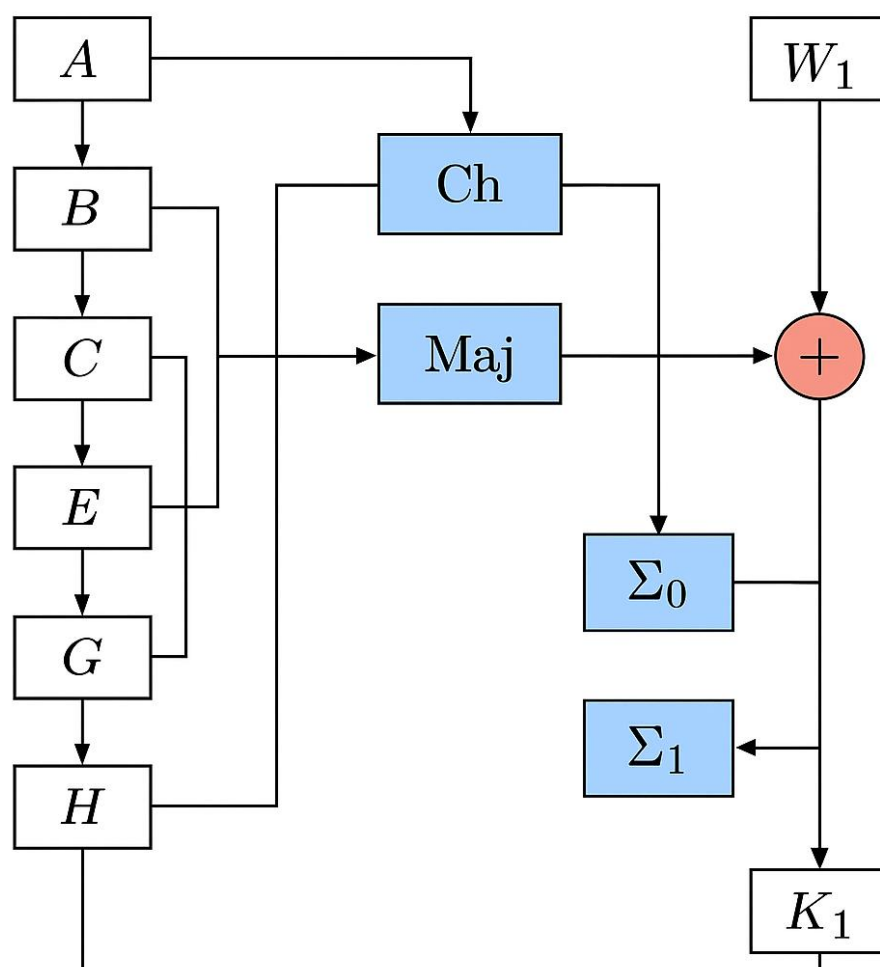


Рисунок 2 – Блок схема компрессионной функции

Каждый раунд принимает одно из 64 подготовленных слов, полученных из 16 слов блока через расширение с помощью константной функции.

После прохождения всех шагов для блока полученные А-Н складываются по модулю 2^{32} с текущим внутренним состоянием. После прохождения всех блоков 256-битное состояние считается выходным хешем функции.

Криптографическая стойкость функции SHA-256 проектировалась с целью избежание коллизий и невозможность нахождения прообраза. На сегодняшний день ни одной коллизии для полного SHA-256 не найдено, во всяком случае неизвестно публично. Данная функция практически исключает коллизии, как случайные, так и преднамеренные, и этот факт отличает ее от 32-битных функций.

SHA-256 проходит множество раундов перемешивания, поэтому даже минимальные различия во входе приводят к абсолютно не коррелированным выходам. Выходные биты SHA ведут себя как независимые случайные при любом изменении входа. С практической точки зрения равномерность распределения у SHA-256 считается отличной. Это означает, что даже для упорядоченных или структурированных данных, таких как JSON, алгоритм SHA-256 формирует равномерно распределённые хеши длиной 256 бит.

За высокую надёжность приходится платить скоростью. Алгоритм SHA-256 существенно сложнее простых хешей – на каждый 512-битный блок совершается 64 раунда с несколькими 32-битными сложениями, логическими операциями и таблицей констант. Иными словами, для обработки каждого байта входных данных алгоритм выполняет десятки операций, что обеспечивает необходимую криптографическую стойкость. В программной реализации SHA-256 может обрабатывать данные со скоростью порядка сотен мегабайт в секунду на современной CPU (без аппаратного ускорения). Если взять для сравнения простой хеш типа Murmur, то он легко превысит гигабайты в секунду на аналогичных ресурсах. Поэтому SHA-256

значительно медленнее некриптографических функций, что, впрочем, ожидаемо из-за разницы в цели. Для небольших сообщений (несколько сотен байтов) задержка SHA-256 может быть пренебрежимо, но при массивной обработке JSON (например, миллионы записей) использование SHA заметно увеличит нагрузку на CPU.

В целом, алгоритм SHA-256 обеспечивает высокий уровень надёжности, однако отличается сравнительно невысокой скоростью работы из-за своей вычислительной сложности.

1.1.2 Криптографический алгоритм ГОСТ Р 34.11-94

Алгоритм хеширования, определённый стандартом ГОСТ Р 34.11-94, – это советско-российская криптографическая хеш-функция, также известная просто как «ГОСТ-хеш». Он был утверждён в 1994 году и до 2013 года являлся обязательным стандартом хеширования в ряде государственных систем РФ. ГОСТ вырабатывает хеш длиной 256 бит (32 байта) из сообщения произвольной длины. Принципиально, ГОСТ Р 34.11-94 похож на другие алгоритмы того же класса (MD5, SHA-1) тем, что реализует схему Меркла–Дамгора, то есть сообщение делится на блоки и проходит через итеративную шаговую функцию, обновляющую внутреннее 256-битное состояние. Однако внутренняя организация шаговой функции у ГОСТ уникальна и основана на национальном стандарте шифрования, в частности, ГОСТ-хеш опирается на симметричный блочный шифр ГОСТ 28147-89 в режиме сжатия. Если говорить кратко, то на каждом раунде хеширования выполняется несколько раундов шифрования данных с использованием частей самого сообщения в качестве ключей.

Алгоритм хеширования ГОСТ Р 34.11-94 также основан на поэтапной обработке входных данных с помощью внутренних преобразований. На первом этапе входное сообщение произвольной длины разбивается на блоки по 256 бит, если длина сообщения не кратна 256 битам, то оно дополняется по определённому правилу. Сначала добавляется сначала один бит «1», затем необходимое число нулей, и в конце добавляется контрольная сумма,

вычисленная как побитовая сумма всех блоков. Это нужно для того, чтобы длина сообщения соответствовала определённому формату, и чтобы итоговое значение хеша зависело не только от самих данных, но и от их размера и структуры.

Затем каждый блок обрабатывается поочерёдно с помощью специальной функции, которая последовательно преобразует их в соответствии с текущим состоянием хеша. В качестве входа она принимает два значения: текущее внутреннее состояние H (256 бит) и очередной блок сообщения M (тоже 256 бит). Данная функция включает в себя три шага.

Первый шаг – генерация четырёх ключей: из текущего состояния H и блока M формируются четыре 256-битных промежуточных ключа K_1 , K_2 , K_3 , K_4 . Ключи формируются по определённому алгоритму, который использует операции побитового исключающего ИЛИ (XOR), циклические сдвиги и прибавление заранее заданных значений. В результате этого процесса входные данные и текущее состояние хеша смешиваются и подготавливаются для следующего этапа обработки.

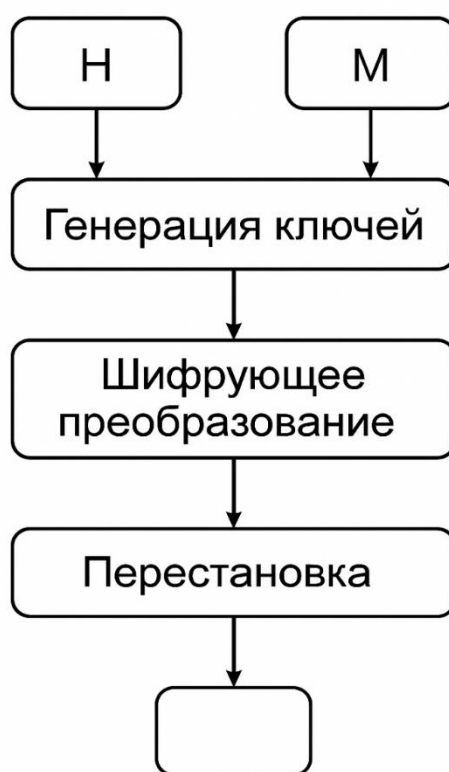
На втором шаге выполняется шифрующее преобразование. Исходное состояние H разбивается на четыре блока по 64 бита, и каждый из них обрабатывается с использованием шифра ГОСТ 28147-89. Это блочный симметричный шифр, состоящий из 32 раундов, применяемый здесь в роли криптографического преобразования. На каждом блоке применяются ключи K_1 – K_4 , полученные на первом шаге, что добавляет зависимости от входного сообщения M . Таким образом, каждый блок H обрабатывается с использованием уникального ключа, который формируется на основе содержимого самого сообщения. Это обеспечивает более надёжное перемешивание данных и повышает стойкость к попыткам предсказания результата

Третий шаг – перестановка и смешивание. К результату шифрования применяется перестановка байтов (так называемая P -преобразование), а затем выполняется операция XOR с исходным сообщением M и предыдущим

значением H . Это делается для усиления лавинного эффекта и обеспечения высокой степени случайности результата. После выполнения этих трёх этапов формируется новое 256-битное значение, которое используется как основа для обработки следующего блока данных.

После того как обработаны все блоки сообщения, в завершение хеш-функция дополнительно обрабатывает общую контрольную сумму всех входных блоков, а также длину исходного сообщения. Это делает результат чувствительным не только к содержимому, но и к количеству и порядку данных. Итогом всей процедуры является 256-битный хеш, который и считается выходом функции.

Рисунок 3 демонстрирует упрощённую структуру шаговой функции алгоритма ГОСТ. В ней показано, как последовательно выполняются три ключевых этапа, такие как формирование ключей, основное шифрование и заключительное перемешивание.



Шаговая функция ГОСТ

Рисунок 3 – Упрощенная схема алгоритма ГОСТ

Последние шаги алгоритма после обработки всех блоков сообщения включают добавление в процесс контрольной суммы всего сообщения и длины сообщения так, как это делает функция SHA. Контрольной суммой является 256-битное суммирование всех блоков без переноса по модулю 2^{256} . Она в конце также пропускается через шаговую функцию, чтобы учесть совокупность всех битов сообщения, данное действие повышает чувствительность хеша к перестановке блоков входа, то есть если блоки поменять местами, общая сумма останется прежней, но порядок обработки нарушится и финальный хеш изменится.

Функция хеширования ГОСТ Р 34.11-94 разрабатывалась как криптографическая и должна обеспечивать надёжную защиту от преднамеренных изменений данных. Алгоритм долгое время считался надёжным и сопоставимым по уровню защиты с зарубежными стандартами, такими как MD5 и SHA-1. Однако по мере развития криптоанализа начали появляться теоретические работы, ставящие под сомнение его полную устойчивость. Так, в 2008 году была описана атака, позволяющая находить коллизии быстрее, чем методом полного перебор, ответ находился примерно за 2^{105} операций. Это, конечно, всё ещё запределный уровень вычислений, недостижимый на практике, но он ниже ожидаемого для идеального 256-битного хеша. Кроме того, предлагались и атаки на первую прообразную стойкость, оцениваемые на уровне 2^{192} операций. Учитывая развитие требований к безопасности и появление новых стандартов, в 2012 году прежний отечественный алгоритм хеширования был заменён обновлённой версией. Несмотря на это, достоверных случаев нахождения коллизий для прежнего алгоритма до сих пор не зафиксировано. Поэтому он по-прежнему может применяться в практических задачах, где не требуется соответствие международным требованиям. Генерация коллизий или подбор другого сообщения с тем же хешем требует фантастических ресурсов и не представляет реальной угрозы в утилитарных приложениях.

Выходные значения хеш-функции ГОСТ распределены равномерно. Такой результат обеспечивается за счёт нескольких этапов внутренней обработки данных, включая многократные раунды шифрования, перестановки байтов и применение специальных таблиц замен, которые в совокупности создают сильный лавинный эффект. Даже малейшие изменения во входных данных приводят к кардинальному изменению результата. С практической точки зрения хеши выглядят как случайные 256-битные числа, вероятность случайного совпадения двух хешей составляет 1 к 2^{256} , что делает такие случаи практически невозможными. Даже если бы база содержала триллион различных JSON-объектов, вероятность того, что два из них получат одинаковый хеш, была бы настолько мала, что её можно считать практически невозможной. Поэтому при использовании алгоритма хеширования из стандарта ГОСТ можно полагаться на то, что каждый JSON будет иметь свой уникальный хеш-отпечаток.

По своей структуре ГОСТ-94 является достаточно сложным алгоритмом. Он использует внутреннюю шифрующую функцию ГОСТ 28147-89, применяя её многократно к каждому блоку данных. В сумме на каждый 256-битный блок выполняется 32 раунда шифрования, плюс дополнительные операции перестановки и сложения. В результате скорость хеширования оказывается ниже, чем у некриптографических функций, но сравнима с SHA-256. На одних платформах ГОСТ-94 может быть немного быстрее, а на других медленнее. Всё зависит от архитектуры и оптимизации реализации.

ГОСТ Р 34.11-94 остаётся надёжной криптографической хеш-функцией, несмотря на появление более новых стандартов. Алгоритм обеспечивает равномерное распределение хеш-кодов, обладает стойкостью к коллизиям и может эффективно применяться для обработки JSON-структур и других типов данных в тех случаях, когда основное внимание уделяется безопасности, а не скорости.

1.2 Некриптографические хеш-функции

В компьютерных науках широко используются и некриптографические хэш-функции, также они называются функциями общего назначения. Их основная цель применения – это эффективная обработка данных, а не криптографическая защита. Такие функции разрабатываются прежде всего с упором на скорость вычисления и хорошие статистические свойства для случайных или типичных входных данных, при условии, что никто намеренно не пытается их сломать [8].

Классическим примером применения некриптографических хеш-функций являются структуры данных, например, для реализаций хеш-таблиц или ассоциативных массивов. В этих структурах хеш от ключа используется для быстрого доступа к элементам по вычисленному индексу. Требуется, чтобы функция равномерно распределяла различные ключи по доступному диапазону значений, минимизируя число коллизий в таблице. Важным свойством здесь является статистическая равномерность, а именно хорошие некриптографические хэш-функции стараются обеспечить максимально равномерное распределение хеш-значений [9]. Это обеспечивает более равномерное распределение элементов по ячейкам таблицы, благодаря чему время поиска остаётся стабильным и предсказуемым.

Кроме того, хеш-функции часто применяются для расчёта контрольных сумм, что позволяет обнаруживать ошибки при передаче или хранении информации. В данном случае хэш-функция служит для проверки целостности данных после передачи, например, если хеш полученного блока не совпадает с хешем исходного, значит данные были искажены. В этом случае от хеш-функции требуется прежде всего чувствительность к изменению данных, чтобы любое искажение заметно влияло на результат, и простота вычисления в ограниченных средах, так как данный тип хеш-функций может использоваться в микроконтроллерах. Криптографическая стойкость в таких задачах, как правило, не требуется, поскольку

предполагается, что ошибки возникают по случайным причинам, а не в результате преднамеренного вмешательства [10].

Некриптографические хеш-функции разрабатываются прежде всего для высокой скорости и равномерного распределения значений, а не для защиты информации. В связи с этим к таким функциям предъявляются особые требования, отличающиеся от требований к криптографическим алгоритмам. Ниже перечислены основные свойства, которые считаются важными для хеш-функций общего назначения.

Одним из главных критериев является высокая скорость работы. Алгоритм должен обрабатываться с минимальными затратами ресурсов, особенно при работе с большими объёмами данных. Для этого некриптографические хеш-функции используют максимально простые операции, например, побитовые сдвиги, сложение, XOR, умножение с переполнением по модулю и т. п. Такие операции выполняются очень быстро, особенно на современных процессорах. Ряд современных алгоритмов, включая MurmurHash и XXHash, разработаны с учётом особенностей процессорной архитектуры и эффективной работы с кэшем. Благодаря этому такие функции часто используются в системах, где важна производительность, например, в базах данных и кэширующих механизмах.

Ещё одним важным свойством считается способность функции выдавать равномерно распределённые хеши по всему диапазону возможных значений. К примеру, при использовании хеш-функции, возвращающей 32-битное значение, предполагается, что для случайных входных данных все возможные 2^{32} хешей распределяются равномерно – без явного преобладания каких-либо значений. Это свойство особенно важно при использовании хешей в хеш-таблицах, так как если одни значения будут встречаться чаще других, произойдёт кластеризация ключей, что приведёт к росту коллизий и снижению эффективности таких структур. Хорошая функция должна сводить к минимуму подобные искажения.

Хорошая хеш-функция должна обладать свойством лавинного эффекта. Это означает, что изменение хотя бы одного бита входных данных должно влиять на все биты выходного значения, то есть очень хорошим результатом является если примерно половина бит хеша изменилась. Это позволяет хеш-функции выдавать существенно отличающиеся значения даже для входов, отличающихся всего одним символом, что особенно важно для повышения устойчивости к коллизиям. В реальности не все некриптографические хеши показывают сильный лавинный эффект. Если брать функцию CRC32, несмотря на свою популярность, она изменяет хеш-последовательность довольно предсказуемо при небольших изменениях входа. Это связано с тем, что алгоритм CRC32 изначально создавался для контроля ошибок при передаче данных, а не для равномерного распределения. В отличие от него, MurmurHash3 обеспечивает хорошее перемешивание и обладает выраженным лавинным эффектом.

Также одно из желаемых свойств функции является наличие явных зависимостей между входными и выходными битами, то есть нельзя, чтобы старшие или младшие биты хеша определялись только частью входа. Когда хеш-функция преимущественно изменяет только младшие биты, оставляя старшие почти без изменений, это говорит о слабом перемешивании. В подобных случаях распределение значений становится неравномерным, что особенно критично при использовании хеша для расчёта индексов в таблицах. Хорошая функция должна обеспечивать, чтобы каждый бит хеша зависел от всех битов входных данных, при этом сильного изменения, как у криптофункций, может не быть, но для поставленных задач оно не является обязательным. Главное, чтобы выходные значения не повторяли закономерностей исходных данных.

Размер хеш-кода у некриптографических функций обычно подбирается под задачу. Чаще всего это 32 или 64 бита. Такого размера обычно бывает достаточно, чтобы коллизии случались крайне редко, но не занималось слишком много памяти. Хеши большей длины снижают риск коллизий, но

вместе с тем увеличивают затраты ресурсов и могут замедлить обработку данных. В редких случаях используются и более короткие, если допустима определённая вероятность совпадений или требуется экономия памяти, при этом стоит помнить, что увеличение длины хеша экспоненциально снижает вероятность коллизии. Проводя расчеты вероятность снижается для 32 бит примерно 1 на 4 миллиарда, для 64 бит она будет уже 1 на 18 квинтильонов.

Ключевым отличием этого класса хеш-функций является то, что они не предназначены для защиты от намеренных атак со стороны злоумышленников и не включают специальных механизмов безопасности. Предполагается, функции не используются для сохранения данных, поэтому не требуется стойкости к преднамеренным атакам. Если вдруг некриптографическая хэш-функция применяется в условиях конфликта с злоумышленником, то возможны проблемы с целостностью данных. Например, известна атака на хеш-таблицы с использованием преднамеренно подобранных коллизий. Злоумышленник генерировал большой набор ключей, которые все дают одинаковый хеш, из-за чего производительность хеш-таблицы резко падала и время работы программы сильно возрастала. Подобная уязвимость характерна для многих старых хэш-функций общего назначения, не рассчитанных на противодействие умышленному подбору коллизий. В ответ на такие угрозы некоторые системы перешли на использование специальных защищённых хэш-функций для структур данных, которые вводят секретный ключ или соль, затрудняя подбор коллизий злоумышленником.

В целом, некриптографические хеш-функции разрабатываются с приоритетом на высокую производительность и равномерное распределение значений, но они не обеспечивают необходимого уровня защиты для использования в криптографии. Их применяют там, где требуется быстро и эффективно сопоставлять данные фиксированным кодам.

1.2.1 Базовая хеш-функция в Java

В языке Java используется метод `hashCode()` класса `java.lang.Object`. Данная функция используется для получения хеш-кода любого объекта. По контракту Java, равные по методу сравнения объекты должны иметь одинаковый хеш-код, однако для двух разных объектов совпадение хешей допускается. В базовом классе `Object` реализация `hashCode()` не основывается на содержимом объекта, а обычно связана с идентичностью объекта в памяти, хотя в других версиях Java может использоваться и информация в объекте [11]. Исторически часто использовался адрес объекта в памяти, преобразованный в целое, однако современная JVM переносит объекты в памяти, поэтому фактическая реализация сложнее. В HotSpot JVM, например, при первом вызове хеш-функции для объекта генерируется псевдослучайное 32-битное число, которое сохраняется в заголовке объекта и возвращается при всех последующих вызовах. Таким образом, базовый `hashCode()` выступает как некое идентификационное число объекта на время жизни в JVM.

Результатом вызова `hashCode()` является целое число типа `int`, то есть 32-битное значение. Однако важно понимать, что по умолчанию это значение не является стабильным между разными запусками программы. Алгоритм может использовать внутренние детали работы виртуальной машины, включая адрес объекта в памяти или начальные значения случайных генераторов. В результате, один и тот же объект при повторных запусках может получить разные хеш-коды, даже если его содержимое остаётся неизменным. Это делает стандартную реализацию `Object.hashCode()` непригодной для долговременной идентификации объектов, особенно если речь идёт о сохранении данных между сессиями или обмене между системами. Наглядная схема вычисления функции представлена на рисунке 4.



Рисунок 4 – Схема вычисления хеш-функции в Java

Поскольку вычисление хеша строится на внутренних механизмах JVM, распределение значений по диапазону `int` в среднем близко к равномерному. Коллизии, при которых два разных объекта получают одинаковое хеш-

значение, допускаются стандартом и считаются допустимыми. Спецификация Java не требует уникальности хешей, поэтому реализация `hashCode()` стремится минимизировать, но не исключить совпадения. Для небольшого количества объектов вероятность коллизий крайне мала, однако по принципу дня рождения она быстро растёт с увеличением количества элементов. Например, при размещении десятков миллионов объектов в хеш-таблице можно практически гарантировать наличие коллизий. В то же время, для большинства повседневных задач этого ограничения не критичны, особенно при работе с ограниченным объёмом данных. Скорость работы метода `hashCode()` – одно из его главных преимуществ. Это очень лёгкая операция, которая, как правило, выполняется мгновенно. При первом вызове хеш может быть вычислен и сохранён внутри объекта, после чего повторные обращения просто возвращают уже готовое значение. Благодаря этому `hashCode()` идеально подходит для использования в хеш-структурах, где требуется быстрое сравнение и поиск по ключу. Среди всех способов получения хешей в Java, базовый `Object.hashCode()` – один из самых быстрых по времени выполнения.

Хотя метод реализован во всех объектах по умолчанию, его поведение не учитывает содержимое объекта. Два экземпляра одного класса с идентичными полями, но разными ссылками в памяти, будут иметь разное поведение как при сравнении, так и у метода `hashCode()`, если эти методы не переопределены. В случаях, когда требуется логическое сравнение объектов по содержимому, программисту необходимо вручную переопределить метод `hashCode()`. Это особенно актуально для таких структур, как JSON-объекты или классы с явно выраженными полями. Без переопределения все экземпляры будут восприниматься как разные, даже если они содержат одни и те же данные.

В случае хранения JSON-объекта как экземпляра класса с соответствующими полями, и при использовании стандартного `hashCode()`, каждый объект будет получать уникальное значение, даже если его поля

полностью совпадают с другим. Это объясняется тем, что по умолчанию сравниваются именно ссылки на объекты, а не их содержимое. Такой подход делает невозможным определение дубликатов на основе данных.

Таким образом базовая функция вычисления хеш-кода в языке Java является примитивной, но быстрой реализацией. Она делает уникальными объекты по их месту в памяти или случайному маркеру, не учитывая содержимое. Для задачи хеширования конкретных данных (например, JSON-объектов по значению полей) требуется использовать другие функции, основанные на информации объекта, либо специальные алгоритмы вроде описанных далее FNV или Murmur.

1.2.2 Хеш-функция FNV-1a

FNV-1a (Fowler–Noll–Vo) – это один из наиболее известных и широко применяемых алгоритмов хеширования общего назначения. Он принадлежит к семейству быстрых и простых в реализации хеш-функций, предназначенных для обработки больших объёмов данных с минимальными вычислительными затратами [12]. На практике FNV-1a используется в таких задачах, как построение хеш-таблиц, индексация, фильтрация дубликатов, генерация уникальных идентификаторов, сериализация и многое другое.

Основная идея алгоритма заключается в последовательной обработке каждого байта входных данных с помощью двух простых операций – логического исключающего ИЛИ (XOR) и умножения на заранее определённую константу. На первом этапе вычисления инициализируется переменная `hash` некоторым фиксированным значением, которое называется начальным смещением (`offset basis`). Это значение отличается для различных битных версий функции (32-битной, 64-битной и т. д.) и подбирается так, чтобы минимизировать число коллизий при характерных входных данных. Затем каждый байт входной строки обрабатывается в цикле. На каждой итерации байт данных вводится в текущее состояние хеша через операцию XOR, после чего результат умножается на специальное простое число. Эта константа известна как `FNV_prime` и тоже различается в

зависимости от разрядности хеша. В результате таких шагов формируется итоговое значение хеша, которое зависит от каждого байта входной строки и хорошо распределяется по диапазону допустимых чисел [13].

Алгоритм FNV-1a является модификацией более раннего варианта – FNV-1. Главное отличие между ними заключается в порядке операций: в FNV-1 умножение выполняется раньше XOR, а в FNV-1a наоборот – сначала выполняется XOR, а затем умножение. Несмотря на схожесть, поведение этих версий на практике отличается. Исследования и эмпирические тесты показали, что именно порядок операций в FNV-1a даёт лучшее распределение хешей, особенно на коротких строках и однотипных шаблонах. Именно поэтому FNV-1a постепенно вытеснил исходную версию и стал де-факто стандартом при реализации FNV. Одним из главных критериев качества хеш-функции является её способность равномерно распределять значения по всему диапазону выходных кодов. В этом плане FNV-1a демонстрирует хорошие результаты. Даже при небольшом изменении входных данных хеш-код меняется значительно, что обеспечивает лавинный эффект. Например, если изменить всего один символ в строке, итоговый хеш будет принципиально иным. Это свойство позволяет эффективно использовать FNV-1a для поиска, сравнения и фильтрации данных.

Визуализация распределения хешей от FNV-1a на последовательностях строк, представляющих, например, числовые значения, представленная на рисунке 5, показывает, что заполнение хеш-таблицы по индексам получается равномерным. В отличие от более простой версии FNV-1, которая при таких входах может образовывать заметные «пустоты» и регулярные узоры в таблице, FNV-1a даёт почти случайное распределение, что повышает эффективность структур данных, использующих хеши.

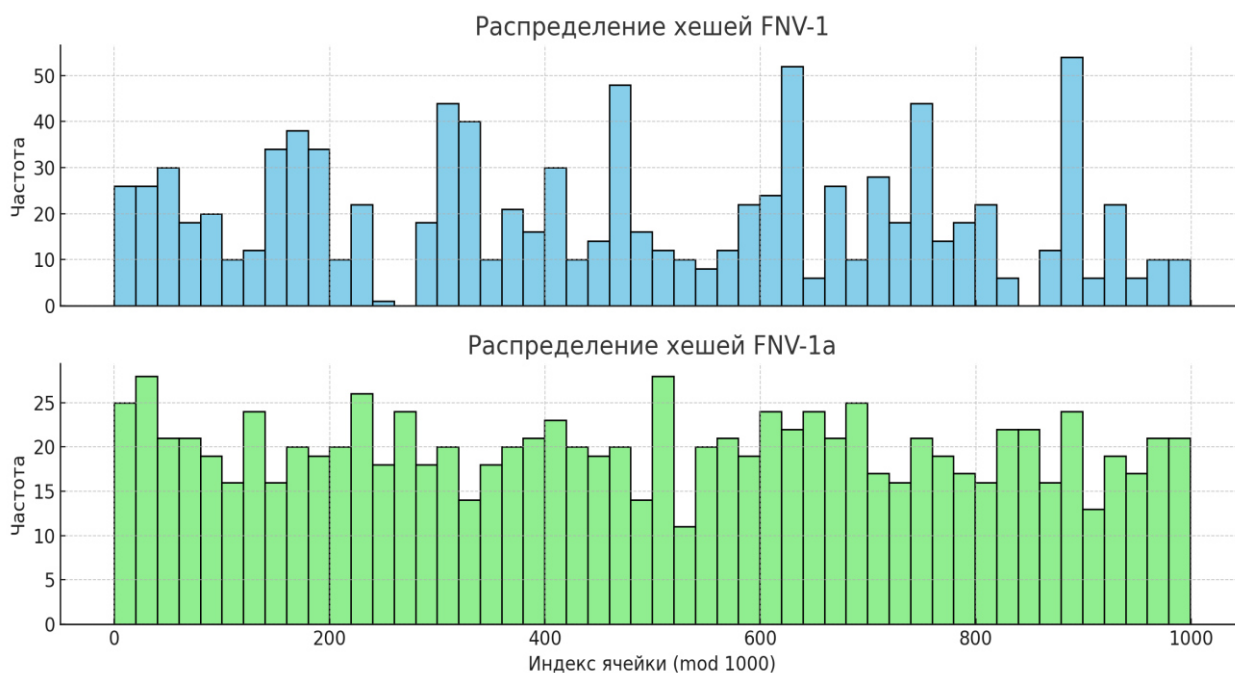


Рисунок 5 - Визуализация распределения

Теоретическая вероятность случайной коллизии в FNV-1a для двух произвольных значений составляет приблизительно 1 к 4 миллиардам. На практике FNV-1a показывает весьма низкий уровень коллизий. В одном из тестов, проведённом на выборке более 200 тысяч английских слов, было зафиксировано всего 4 случая совпадений хешей. Это сопоставимо с другими быстрыми хеш-функциями общего назначения, такими как MurmurHash или DJB2, что говорит о высокой статистической устойчивости алгоритма. Следует, однако, помнить, что FNV-1a не обладает криптографической стойкостью, так как при специально подобранных данных можно искусственно вызвать коллизию. Поэтому применять его в целях безопасности не рекомендуется.

Одним из основных достоинств FNV-1a является его высокая скорость. Алгоритм выполняет всего две лёгкие операции над каждым байтом данных, такие как XOR и умножение. Эти операции не требуют доступа к памяти, не зависят от сложных таблиц, не используют условные переходы и идеально подходят для выполнения на современных процессорах. Благодаря этому FNV-1a может обрабатывать большие массивы данных практически «на лету», не создавая значительной нагрузки на систему.

В различных тестах среднее время хеширования одного слова при использовании FNV-1a составляло порядка 150 наносекунд, что сопоставимо с одним из самых быстрых алгоритмов. Это делает FNV-1a хорошим выбором для приложений, где требуется быстрое хеширование большого количества строк или объектов, особенно в потоковых системах или при построении индексов.

Алгоритм FNV-1a отлично справляется с задачами хеширования JSON-объектов. Один из простейших подходов заключается в сериализации объекта в строку в формате JSON, а затем применении FNV-1a ко всей строке. Такой способ позволяет получать компактный хеш, однозначно отражающий содержимое объекта. Это может использоваться, например, для определения изменений в структуре, обнаружения дубликатов, создания быстрых ключей для поиска и кэширования.

Благодаря равномерному распределению и чувствительности к изменениям, даже незначительное изменение в JSON (например, замена одного значения поля) приведёт к совершенно новому хешу. Таким образом, FNV-1a можно использовать в качестве основы для построения эффективных и надёжных индексов, особенно если важна скорость обработки данных, а не защита от атак.

Подводя итог, алгоритм FNV-1a представляет собой удачное сочетание простоты, скорости и достаточной надёжности для прикладных задач. Он хорошо зарекомендовал себя при хешировании строк, структурированных текстов и сериализованных объектов, в том числе JSON.

1.2.3 Хеш-функция MurmurHash3

MurmurHash3 – это хеш-функция общего назначения, которая не относится к классу криптографических, но при этом обеспечивает высокую скорость и хорошее равномерное распределение значений [14]. Она была разработана Остином Эплби (Austin Appleby) примерно в 2011 году как третья версия семейства Murmur, и до сих пор активно используется в практике программирования.

Функция MurmurHash3 существует в двух основных вариантах, таких как с 32-битным и 128-битным выходом. Вариант с 32-битной версией кроссплатформенен и может использоваться на любых архитектурах, тогда как 128-битная имеет подверсии, оптимизированные под 32- и 64-битные процессоры. В рамках этого исследования основное внимание будет уделено 32-битной реализации, поскольку она ближе по структуре и назначению к другим широко применяемым хешам, таким как FNV-1a.

В ходе работы алгоритма MurmurHash3 входные данные обрабатываются поэтапно, где каждый этап играет важную роль в обеспечении лавинного эффекта и равномерного распределения выходных значений.

На первом этапе входная строка разбивается на последовательность блоков по четыре байта. Каждый из этих блоков интерпретируется как целое число. После этого выполняется начальное преобразование. Оно включает в себя умножение блока на заранее определённую константу, которая подобрана так, чтобы результат максимально изменялся даже при незначительном отличии входных данных. Эта операция позволяет усилить чувствительность алгоритма к каждому байту [15].

Далее полученное значение подвергается циклическому сдвигу влево на определённое количество битов. Циклический сдвиг – это операция, при которой старшие биты, ушедшие за пределы разрядности, возвращаются в младшие разряды. Таким образом достигается дополнительное перемешивание данных внутри блока.

После сдвига выполняется второе умножение результат на другую константу, также заранее зафиксированную. Эта операция ещё сильнее изменяет внутреннее представление блока, способствуя распределению изменений по всем битам.

Полученный результат объединяется с текущим значением хеша с помощью операции XOR. Это значит, что каждый бит результата влияет на соответствующий бит хеша. Чтобы избежать накопления закономерностей,

алгоритм дополнительно выполняет сдвиг и арифметическую операцию над самим хешем, снова используя константы. В итоге, даже если какие-то биты остались не полностью перемешанными, они не останутся изолированными в финальном результате.

Если длина входных данных не делится на четыре, то оставшиеся байты также учитываются. Они объединяются в вспомогательное значение и проходят тот же набор преобразований, хотя и в немного сокращённой форме.

В завершение алгоритм применяет серию дополнительных шагов, таких как завершающее перемешивание, известное как финальная трансформация. Оно включает несколько XOR-операций и умножений, направленных на усиление хаотичности и непредсказуемости результата. Этот этап особенно важен, чтобы устранить возможные корреляции между входом и выходом и гарантировать, что даже малейшее изменение входных данных даст совершенно иной хеш-код.

Одним из основных достоинств MurmurHash3 считается его равномерное распределение значений. Даже на предсказуемых наборах данных, вроде последовательностей чисел или слов с повторяющимися частями, хеши распределяются по диапазону почти как случайные. Это достигается за счёт тщательно подобранных операций перемешивания и использования нестандартных констант. Визуализация распределения значений показывает, что заполнение хеш-таблицы при использовании Murmur не имеет выраженных «пустых» зон, а значения кажутся случайными и непредсказуемыми. На рисунке 6 представлена упрощённая блок-схема функции.



Рисунок 6 – Блок схема алгоритма

Коллизии при использовании MurmurHash3 возникают довольно редко. Теоретически, как и у любой 32-битной функции, вероятность случайного совпадения хешей у двух произвольных объектов составляет 1 к 4 миллиардам. На практике, при тестировании на наборах из сотен тысяч уникальных строк наблюдается всего несколько коллизий. Это сопоставимо с результатами других хороших некриптографических хеш-функций и подтверждает, что Murmur надёжен в реальных задачах.

MurmurHash3 спроектирован таким образом, чтобы эффективно использовать возможности современных процессоров. Алгоритм работает с 4-байтовыми блоками, что соответствует размеру большинства машинных слов. Операции умножения, сдвига и XOR хорошо оптимизируются процессором, особенно при использовании конвейеров команд и инструкций SIMD. Благодаря этому Murmur может обрабатывать большие объёмы данных с очень высокой скоростью, в некоторых случаях быстрее, чем другие функции.

Сравнительные тесты показывают, что на коротких строках время хеширования одного элемента составляет примерно 100–150 наносекунд, что делает Murmur одним из самых быстрых алгоритмов в своей категории. MurmurHash3 особенно хорошо показывает себя при хешировании структурированных данных, таких как JSON. Один из типичных способов применения – это сериализация JSON-объекта в строку и вычисление хеша от этой строки. Это позволяет получить компактное и стабильное значение, пригодное для сравнения, индексации или кэширования.

Таким образом, рассмотренная хеш-функция является хорошим вариантом для общего назначения. Она сочетает в себе отличную скорость, равномерное распределение, низкий уровень коллизий и широкую доступность. Несмотря на отсутствие криптографической стойкости, её можно безопасно использовать во всех задачах, где не требуется защита от атак. В приложениях, работающих с JSON и другими структурированными данными, Murmur обеспечивает быструю и надёжную генерацию хешей, минимизирует вероятность дубликатов и легко масштабируется.

1.2.4 Хеш-функция CRC32

CRC32 (циклический избыточный код) – это одна из самых старых и надёжных хеш-функций, которая изначально разрабатывалась вовсе не для хеширования в привычном нам смысле, а для обнаружения ошибок при передаче или хранении данных. Тем не менее, благодаря своей простоте, скорости и определённому уровню надёжности, она широко применяется в

самых разных сферах, включая хеш-таблицы, файловые системы и сетевые протоколы.

Основная идея CRC32 строится на математике полиномов над конечным полем $GF(2)$, где все операции выполняются по модулю 2, то есть фактически с использованием побитовой арифметики. В терминах хеширования можно представить CRC32 как процесс деления длинной двоичной строки, представляющей входные данные, на заранее определённый полином. Результатом этого деления становится остаток, который и служит значением хеш-функции.

Представленный полином в шестнадцатеричном виде соответствует маске 0x04C11DB7, однако на практике в современных реализациях CRC32 вместо длинного деления используется гораздо более удобный алгоритм на базе побитовых сдвигов и операций XOR. На начальном этапе создаётся регистр длиной 32 бита, инициализированный, как правило, значением 0xFFFFFFFF. Затем для каждого байта входных данных происходит обработка. Используется битовая маска и комбинируется с текущим значением регистра, и на основе таблицы или формулы происходит его обновление. В конце вычислений результат инвертируется, полученное значение считается окончательным хешем. В реализации могут использоваться разные подходы. Начиная от побитового вычисления до более производительных вариантов, в которых применяется таблица из 256 предвычисленных значений. Кроме того, в современных процессорах, например, в линейке Intel, имеются аппаратные инструкции, позволяющие выполнять CRC32 прямо на уровне процессора за один такт. Блок схема, иллюстрирующая этапы функции, представлена на рисунке 7.

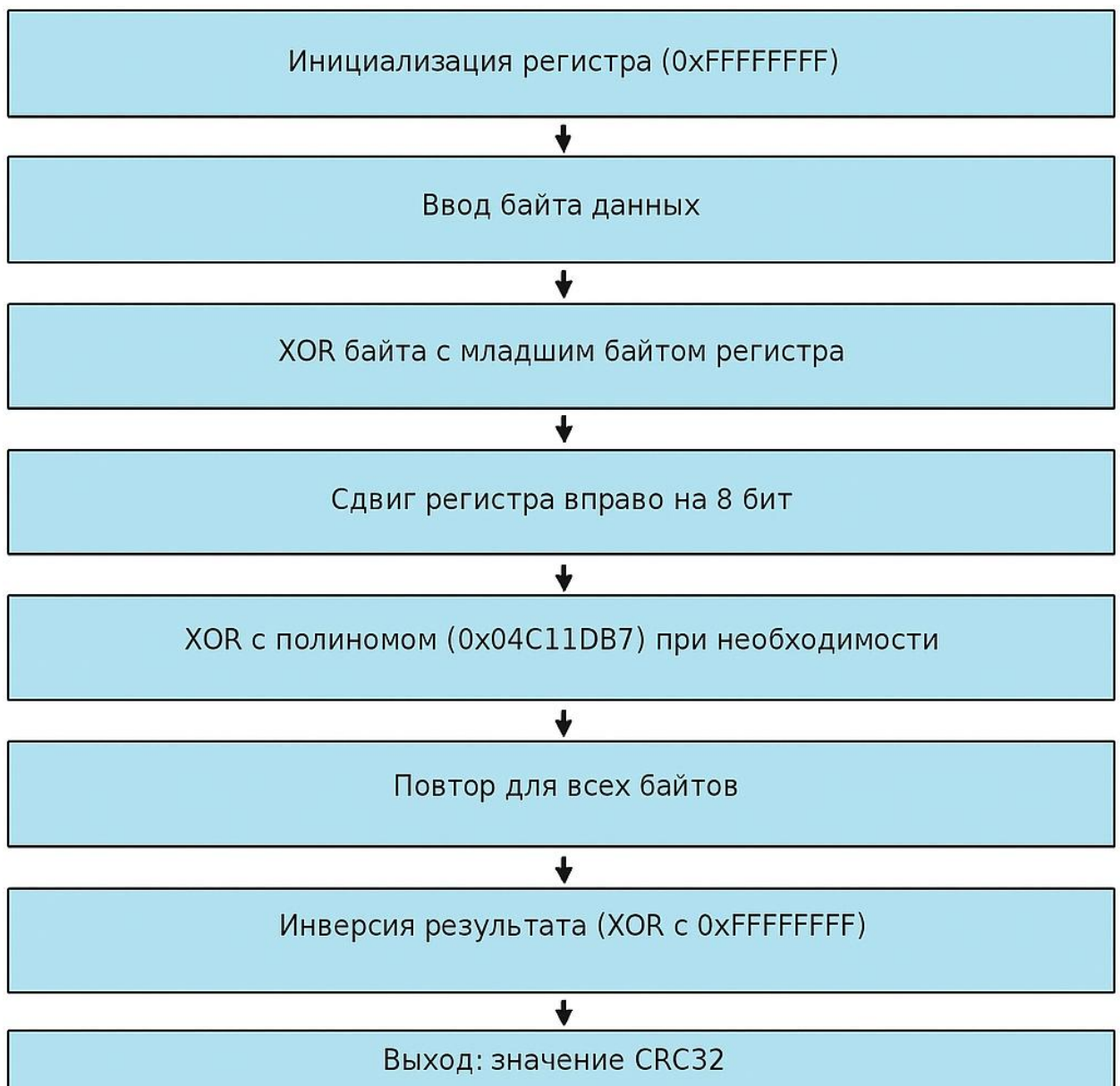


Рисунок 7 – Блок схема функции

CRC32 не создавалась как универсальная хеш-функция. Первоначальной задачей было быстрое обнаружение случайных ошибок, возникающих, например, при передаче данных по сетям или при чтении с диска. Тем не менее, по сути, CRC32 даёт 32-битное числовое представление данных, и его вполне можно рассматривать как хеш-значение. Одной из отличительных черт CRC32 является её линейность. Это означает, что если известны два разных блока данных и их контрольные суммы, то можно легко рассчитать CRC от их комбинации, используя лишь операции XOR. Однако именно эта предсказуемость делает CRC уязвимой для криптоанализа, зная,

как вычисляется хеш, злоумышленник может сгенерировать данные, дающие тот же результат. Поэтому для задач, связанных с безопасностью, CRC32 не применяют.

Несмотря на ограниченность с точки зрения криптостойкости, CRC32 довольно хорошо справляется со своей задачей в прикладных ситуациях, если данные разнообразны и не специально подобраны для атаки, распределение хешей получается близким к равномерному. Для случайных входов вероятность коллизии такая же, как и у любой другой 32-битной функции: примерно 1 к 4 миллиардам.

В различных тестах CRC32 показывает низкое количество коллизий. Например, при хешировании более 200 тысяч GUIDов не было зафиксировано ни одного совпадения. В аналогичных тестах на английских словах было всего два случая коллизий. Это позволяет утверждать, что при случайных входных данных CRC32 работает стабильно и надёжно, по уровню равномерности и коллизий находясь примерно на одном уровне с такими функциями, как FNV или MurmurHash.

Однако стоит иметь в виду, что равномерность может нарушаться, если данные имеют специфическую структуру, особенно если отличаются только перестановкой битов, но при работе с обычными текстовыми данными таких проблем не возникает.

Реализация CRC32 не представляет особой сложности. В большинстве языков программирования существуют готовые функции для её вычисления. В Java, например, CRC32 реализована в стандартной библиотеке `java.util.zip`, и для её использования достаточно создать объект нужного класса и скормить ему байты, но если же требуется своя реализация, например, для особых параметров, то можно использовать побайтовый метод с таблицей или реализовать более универсальный побитовый вариант. Главное стоит быть внимательным к параметрам, так как существует несколько вариантов CRC32, отличающихся нюансами.

CRC32 является достаточно быстрой функцией. Алгоритм предельно лёгкий с точки зрения операций. Основное время занимает побитовая обработка и работа с таблицей, которая занимает всего 1 килобайт памяти. При использовании аппаратных инструкций скорость выполнения возрастает до нескольких гигабайт в секунду.

CRC32 может применяться и для хеширования структурированных данных, таких как JSON. Самым распространённым сценарием является сериализация объекта в строку и вычисление CRC по её байтовому представлению. Полученный хеш можно использовать для контроля целостности или быстрой проверки изменений. Например, при кэшировании можно сохранить вместе с JSON его CRC и при следующем вызове быстро проверить, не изменилась ли структура данных.

Подводя итог, CRC32 является проверенной временем хеш-функция, которая отличается простотой реализации и высокой скоростью выполнения. Она обеспечивает равномерное распределение значений при случайных входных данных и показывает очень низкий уровень коллизий.

1.3 Сравнительный анализ хеш-функций

В рамках данного исследования были рассмотрены шесть различных хеш-функций, различающихся как по внутреннему устройству, так и по применимости в задачах прикладного и криптографического характера. Сравнение охватывает следующие алгоритмы: SHA-256, ГОСТ Р 34.11-94, FNV-1a, MurmurHash3, CRC32 и стандартный `Object.hashCode()` в Java.

Сравнительный анализ представлен на таблице 1.

Таблица 1 –
Сравнительный
анализ

Алгоритм	Криптостойкость	Равномерность	Вероятность коллизии	Скорость	Сложность реализации	Подходит для JSON
SHA-256	Высокая	Отличная	Незначительная (2^{128})	Низкая	Высокая	Да (контроль изменений)
ГОСТ Р 34.11-94	Умеренно высокая	Хорошая	Теоретически снижена (2^{105})	Низкая	Высокая	Да (в гос. стандартах)
FNV-1a	Отсутствует	Средняя	Высокая при больших объёмах	Очень высокая	Очень низкая	Ограниченно
MurmurHash3	Отсутствует	Отличная	Низкая для непреднамеренных данных	Очень высокая	Средняя	Да (сравнение)
CRC32	Отсутствует	Хорошая	Низкая случайно, высокая при атаке	Очень высокая	Низкая	Ограниченно (целостность)
Object.hashCode	Отсутствует	Псевдослучайная	Зависит от реализации, возможны	Максимальная	Не требуется вручную	Нет (для сравнения содержимого)

С точки зрения криптостойкости, наивысший уровень демонстрирует SHA-256. Этот алгоритм проектировался с учётом требований безопасности и до сих пор не имеет известных практических уязвимостей. ГОСТ R 34.11-94 также позиционировался как криптографически стойкий, но исследования последних лет выявили потенциальные способы теоретического подбора коллизий, что снизило доверие к нему. Остальные алгоритмы к криптографическим не относятся и не обеспечивают устойчивость к атакам.

По равномерности распределения выходных значений лидируют SHA-256 и MurmurHash3. Оба алгоритма дают практически случайное распределение хешей при любых входных данных. FNV-1a работает удовлетворительно, но на последовательных шаблонах может демонстрировать закономерности. CRC32 также показывает хорошее поведение, особенно если входные данные имеют хаотичную природу. `Object.hashCode()` в своей базовой реализации не предназначен для равномерного распределения – его результат зависит от внутреннего состояния JVM, что делает его непредсказуемым при повторном запуске и слабо применимым для анализа по содержимому объекта.

Вероятность коллизий зависит напрямую от длины хеша и особенностей реализации. SHA-256 практически исключает коллизии в прикладных задачах. Вероятность совпадения хешей двух различных входов можно считать нулевой. ГОСТ-94 немного уступает по стойкости, но в практических условиях тоже даёт надёжный результат. Некриптографические алгоритмы обеспечивают низкий уровень коллизий при случайных данных, однако не защищены от намеренного подбора. `Object.hashCode`, в свою очередь, не гарантирует уникальности даже в рамках одного запуска, если не переопределён вручную.

Одним из решающих факторов является скорость выполнения при выборе хеш-функции для массовых вычислений. Наиболее быстрые алгоритмы – это FNV-1a, MurmurHash3 и CRC32. Они работают почти мгновенно, обрабатывая данные с минимальными накладными расходами.

SHA-256 и ГОСТ существенно медленнее из-за своей криптографической природы: большое количество раундов, использование сложных битовых операций и постоянные обращения к таблицам замедляют вычисление. `Object.hashCode()` вычисляет хеш-код быстрее других, так как возвращает готовое значение без перерасчёта, но ценой полной недетерминированности и слабой смысловой ценности.

Сложность реализации также варьируется. Простейшим в реализации является FNV-1a, он использует несколько строчек кода, не зависящих от внешних библиотек. CRC32 немного сложнее, особенно если писать вручную без таблицы, но тоже хорошо изучена. Murmur требует внимания к константам и порядку операций, особенно если нужна совместимость с другими платформами. SHA-256 и ГОСТ являются наиболее сложные в ручной реализации из-за многоблочной структуры, многочисленных раундов и сложных битовых преобразований. Метод `hashCode()` реализован по умолчанию в каждом объекте Java и не требует вмешательства, но его поведение редко соответствует прикладным требованиям.

Применимость к JSON-данным во многом зависит от целей. Для задач, связанных с безопасностью и контролем целостности, лучше всего подойдут SHA-256 или ГОСТ. Если требуется быстрый идентификатор содержимого объекта (например, для кэширования), удобны MurmurHash3 или FNV-1a. CRC32 может использоваться в качестве контрольной суммы или базового способа проверки изменений, хотя криптографической защиты не обеспечивает. `Object.hashCode()` нельзя использовать для сравнения JSON по содержимому, так как он работает с ссылками, а не с данными.

2 Выбор средств разработки

В данном разделе рассматривается набор технологий и инструментов, выбранных для разработки веб-приложения, предназначенного для анализа JSON-объектов и интеллектуального подбора оптимальной хеш-функции с помощью модели искусственного интеллекта. Такой проект включает бэкенд-сервис на Java, модуль ИИ на Python, веб-интерфейс на JavaScript, базу данных и сопутствующие инструменты. Ниже подробно описаны причины выбора каждого ключевого средства разработки, а также то, как они взаимодействуют друг с другом в рамках системы [16].

2.1 Выбор технологий для backend части

Для разработки программного аппаратной части приложения был выбран язык Java. Этот язык был выбран в качестве основного языка для серверной части (бэкенда) благодаря своей надежности, высокой производительности и богатой экосистеме для корпоративной разработки. Java является статически типизированным языком, что помогает выявлять многие ошибки на этапе компиляции. Экосистема Java славится множеством готовых библиотек и решений, что особенно важно для задач работы с хеш-функциями и веб-сервисами.

Для построения RESTful API на Java был использован фреймворк Spring Boot. Spring Boot значительно упрощает создание веб-служб за счёт автоконфигурации и удобных технологий для типовых задач. Разработчики всё чаще предпочитают Spring Boot вместо первоначального Spring, поскольку он гораздо проще в начальной настройке, быстрее в работе и облегчает развертывание приложений [17]. Для разработки программного обеспечения позволяет быстро создать REST API без необходимости вручную настраивать сервер, благодаря встроенному web-серверу [18]. Наличие встроенного сервера означает, что приложение можно запускать локально как самостоятельный JAR-файл, не устанавливая отдельно контейнер сервлетов, что значительно ускорит цикл разработки и

тестирования. Кроме Tomcat, при желании можно переключиться и на другие встроенные серверы, но в рамках проекта необходимости в этом нет.

Ещё одной причиной выбора Spring Boot стала его интеграция с другими компонентами и библиотеками. Фреймворк предоставляет множество готовых модулей, упрощающих решение распространённых задач. В контексте разрабатываемого приложения особенно ценным будет модуль Spring Data JDBC для удобной работы с базой данных PostgreSQL. Spring Boot автоматически настроит подключение к PostgreSQL через зависимость `spring-boot-starter-data-jdbc`, избавив от ручной конфигурации драйверов. В целом, Spring Boot обеспечивает быструю и безопасную настройку среды разработки, поэтому можно сразу приступить к написанию бизнес-логики, не тратя много времени на шаблонный код конфигурации. Также за счёт использования аннотаций и умных соглашений Spring Boot сокращает объем кода и делает его более понятным, тем самым разработчик фокусируется на функциональности. Всё это делает Spring Boot оптимальным выбором для реализации REST API в дипломном проекте.

Наконец, широкое распространение Spring Boot означает наличие большого сообщества и обилия документации. В случае затруднений всегда можно найти примеры и документацию, что существенно снизит риски при разработке дипломной работы. Таким образом, связка Java и Spring Boot даст устойчивый и проверенный базис для серверной части, обеспечив производительность, масштабируемость и удобство разработки.

2.2 Модуль для искусственного интеллекта

Для разработки модуля искусственного интеллекта, который анализирует JSON-объекты и помогает подбирать оптимальную хеш-функцию, был выбран язык Python. Это обосновано тем, что Python де-факто стал стандартом индустрии для задач машинного обучения и анализа данных. Он обладает богатой экосистемой библиотек для ИИ, которые сильно ускоряют разработку модели. В частности, в проекте будут использоваться Scikit-learn, Pandas и NumPy.

Scikit-learn – одна из самых популярных библиотек машинного обучения в Python. Она предоставляет простой и эффективный инструментарий для построения и оценки моделей, включая множество реализованных алгоритмов. Использование Scikit-learn позволит не вдаваться в подробности алгоритмов, а позволит воспользоваться готовыми реализациями алгоритмов при создании модели, которая рекомендует хеш-функцию. Вместе с Scikit-learn применялась библиотека Pandas для подготовки и анализа данных. Pandas удобна для работы с табличными данными и умеет загружать JSON-структуры, преобразуя их при необходимости в табличный вид. Это будет полезно при извлечении признаков из JSON-объектов, так как можно будет собирать статистику по структуре и содержимому JSON, которую затем будет воспринимать модель ИИ. NumPy, лежащая в основе Pandas и Scikit-learn, обеспечит эффективные низкоуровневые операции с массивами и матрицами. В совокупности эти библиотеки составляют мощный стек для быстрой разработки и экспериментов с моделью машинного обучения.

Отдельно стоит отметить, что можно рассматривалась возможность применения библиотек для глубокого обучения, таких как TensorFlow или PyTorch, если задача потребует более сложной нейронной сети. Однако для цели дипломной работы хватит возможностей Scikit-learn. Тем не менее, сам факт использования Python оставляет простор для будущего улучшения модели. В дальнейшем при необходимости интеграции нейросети можно подключить TensorFlow, благо Python позволяет это сделать без смены экосистемы.

Для интеграции Python-модуля ИИ с основным приложением будет реализован отдельный REST-сервис с использованием микро-фреймворка Flask. Flask был выбран благодаря своей простоте и легкости: это минималистичный фреймворк, позволяющий буквально в несколько строк кода создать веб-сервис для вызова модели. Flask предоставляет все необходимое для быстрого написания REST API с минимальными

накладными расходами на конфигурирование [19]. В случае дипломной работы Flask-сервис будет предоставлять конечную точку, принимающую на вход JSON и возвращающую рекомендованную хеш-функцию, и метрику качества. Такая архитектура поможет разделить ответственность, ведь Java-приложение обращается к Python-сервису через HTTP, что упрощает отладку и даёт возможность масштабировать каждый компонент независимо [20].

Стоит упомянуть что FastAPI является современным асинхронным фреймворк на Python, который также подходит для создания REST-сервисов. FastAPI обладает рядом преимуществ, например, высокой производительностью и автоматической генерацией документации OpenAPI. Кроме того, FastAPI сразу поддерживает асинхронные запросы, тогда как Flask работает в основном в синхронном режиме. Однако, учитывая относительную простоту нашего сервиса был сделан выбор в пользу Flask.

2.3 Библиотеки хеширования в Java

Поскольку задача проекта связана с вычислением хеш-функций для JSON-объектов, в бэкенде на Java были рассмотрены и использованы различные библиотеки для хеширования. Выбор хеш-функций был важным аспектом, так как планировалось экспериментировать с разными алгоритмами и оценивать их эффективность.

Для криптографических хеш-функций в Java есть встроенное средство, такое как класс MessageDigest из пакета java.security [21]. Он позволяет вычислять хеши с помощью алгоритмов, реализованных в платформе Java, обеспечивая корректность и устойчивость к коллизиям, требуемую в криптографических сценариях. В контексте нашего проекта криптографические хеши могли использоваться как один из вариантов для сравнения с более быстрыми некриптографическими алгоритмами. Однако, поскольку цель дипломной работы оптимально хешировать данные с точки зрения производительности и числа коллизий, больший интерес представляет некриптографические хеш-функции.

К некриптографическим (или прикладным) хеш-функциям относятся алгоритмы, оптимизированные по скорости и распределению хеш-значений, но не предназначенные для криптостойкости. Были выбраны перечисленные выше хеш-функции.

MurmurHash – современный алгоритм, известный высокой скоростью и качественным распределением хешей. Он широко используется в системах хранения и индексирования данных. В проекте MurmurHash интересен тем, что существует в нескольких вариантах (32-битный, 128-битный и др.) и показывает отличные результаты на больших объемах данных. Согласно некоторым исследованиям, реализация MurmurHash3 существенно опережает по скорости классические алгоритмы. Это значимое преимущество для сценария, где нужно хешировать множество JSON-объектов и проверять коллизии, ведь быстрое действие прямо влияет на удобство работы системы.

FNV (Fowler–Noll–Vo) – более старый, но тоже довольно быстрый алгоритм хеширования. Он прост в реализации и обеспечивает неплохое распределение для многих типов данных. Хотя FNV несколько уступает MurmurHash по качеству хеширования и скорости на современных процессорах, мы рассматривали его как базовый некриптографический алгоритм для сравнения. Встроенных средств для FNV в стандартной библиотеке Java нет, но его легко реализовать вручную в несколько строк, поэтому при эксперименте с FNV будет написан самостоятельно на базе описания алгоритма.

CRC32 – это алгоритм контрольной суммы, который также можно использовать как хеш-функцию для не критичных к коллизиям задач. В Java есть готовый класс `java.util.zip.CRC32`, вычисляющий CRC32. Он очень быстрый, так как реализован на низком уровне, однако длина хеша всего 32 бита, что может приводить к частым коллизиям на большом объеме данных. В работе был использован CRC32 в основном для оценки максимальной скорости хеширования, но есть понимание, что по качеству хеша он значительно уступает другим.

Помимо встроенных средств, будут использованы внешние библиотеки для удобства и расширения списка алгоритмов. Одной из них будет Google Guava, содержащая утилитный класс Hashing, предлагающий готовые реализации многих хеш-функций через простой интерфейс. С помощью Guava будет получен доступ к таким функциям, как MD5, SHA-256, CRC32, а также современным некриптографическим алгоритмам MurmurHash3. Использование Guava заметно упростит код, так как она предоставляет сразу готовые хеш-функции.

Также же будет использована библиотека BouncyCastle. Это широко известный криптографический провайдер для Java, добавляющий поддержку множества алгоритмов, отсутствующих в стандартной поставке. В рамках дипломного проекта BouncyCastle пригодится, если потребуются редкие хеш-функции или более гибкая работа с криптографическими примитивами. Хотя непосредственной необходимости в нём не может и не возникнуть.

Подводя итог, выбор библиотек хеширования продиктован стремлением покрыть разные классы алгоритмов. В работе будут задействованы как стандартные средства Java для криптохешей, так и внешние инструменты для быстрых некриптохешей. Это даст возможность в экспериментальной части проекта протестировать несколько подходов хеширования JSON-данных. Впоследствии модель на Python будет использовать результаты этих различных хеш-функций как признаки для обучения и тем самым, разнообразие алгоритмов обогащает пространство признаков для ИИ. Такой подход обеспечивает более объективный выбор оптимальной хеш-функции под конкретные особенности входных данных.

2.4 Выбор технологий для визуальной части приложения

За основу разработки клиентской части приложения была выбрана библиотека React. Это популярная JavaScript-библиотека для создания пользовательских интерфейсов. React был выбран по нескольким причинам. Во-первых, он позволяет относительно быстро и удобно создавать интерактивные одностраничные приложения. Для дипломного проекта

планируется веб-интерфейс, где пользователь может вводить или загружать JSON-объекты, запускать их анализ и получать результаты. Реализовать такую динамическую логику удобнее всего с помощью React, благодаря его декларативному подходу и реактивности [22].

React использует компонентную архитектуру. Приложение разбивается на независимые компоненты, каждый из которых инкапсулирует свою часть интерфейса и логику. Этот принцип значительно упрощает поддержку и масштабирование кода. Если один компонент выходит из строя, это не влияет напрямую на работу других – каждый элемент изолирован в рамках своего состояния [23]. Такая изолированность компонентов повышает стабильность приложения, ведь ошибка в одной части интерфейса не вызывает каскада проблем во всем приложении [24]. Для дипломного проекта это означает, что можно разрабатывать и отлаживать, скажем, компонент загрузки JSON отдельно от компонента вывода результатов, их связь минимальна и проходит через чётко определённые свойства и события.

Ещё одним ключевым преимуществом React является виртуальный DOM и производительность интерфейса. React не манипулирует напрямую реальным DOM браузера при каждом изменении состояния, вместо этого он обновляет виртуальную копию DOM и эффективно вычисляет разницу, применяя только необходимые изменения к реальному интерфейсу. Это даёт высокую скорость работы приложения, особенно заметную при сложных обновлениях UI. Пользователь практически мгновенно видит отклик на свои действия, даже если сама страница довольно тяжеловесна. В дипломной работе, где может происходить визуализация статистики, например, график распределения коллизий разных хеш-функций, и обновление результатов в режиме реального времени, такая эффективность очень важна. React обеспечивает плавность работы интерфейса даже при частых обновлениях данных [25].

React был выбран также благодаря большому сообществу и количеству готовых решений. Эта библиотека на сегодняшний день крайне востребована

– по некоторым оценкам, её используют около 40% фронтенд-разработчиков, что делает React одним из самых популярных инструментов в сфере. Для практически любой задачи можно найти готовый компонент или пример реализации. В частности, для визуализации результатов анализа хеш-функций можно интегрировать сторонние React-библиотеки графиков. Такая экосистема существенно ускоряет разработку дипломного проекта, так как многие элементы не нужно писать с нуля.

Альтернативой React могли быть другие фреймворки, например Angular или Vue.js. Но Angular является более тяжеловесным фреймворком со строгой структурой, избыточный для нашего относительно небольшого приложения, а Vue.js во многом схож с React, но более сложен в освоении [26]. React же хорошо себя зарекомендовал своей гибкостью, позволяя свободно выбирать структуру проекта. Для дипломной работы, где ценится понимание процесса, а не следование канонам фреймворка, React оказался комфортным выбором.

Можно отметить, что использование React потребовало создания backend-for-frontend архитектуры, то есть разработки API на сервере и клиента, потребляющего этот API. Это классический подход для современных веб-приложений. React-приложение в итоге представляет собой одностраничное веб-приложение, общающееся с нашим Java REST API и с Python-сервисом. Такой отдельный фронтенд облегчил отладку.

В итоге React как средство разработки фронтенда был выбран за современность, удобство и богатый набор возможностей для построения UI. Студенту-разработчику эта библиотека позволила применить на практике знания, полученные в учебных проектах, и создать приятный и отзывчивый интерфейс для демонстрации работы интеллектуальной системы выбора хеш-функций.

2.5 Выбор базы данных

В качестве системы управления базами данных для проекта выбрана PostgreSQL. Она будет использоваться для хранения JSON-объектов,

загруженных пользователями, а также для результатов вычисления хешей, рекомендаций модели, истории запросов и пр. PostgreSQL – это реляционная СУБД с открытым исходным кодом, известная своей надежностью и богатым функционалом. Выбор PostgreSQL обусловлен несколькими важными преимуществами.

Во-первых, поддержка JSON. PostgreSQL одной из первых внедрила полноценную поддержку JSON-формата на уровне базы данных. Начиная с версии 9.4, в PostgreSQL есть двоичный тип данных JSONB (Binary JSON), который идеально подходит для хранения полуструктурированных данных, таких как объекты JSON. JSONB хранит данные в бинарном виде и допускает вложенные структуры, массивы и т.д., при этом обеспечивая эффективное хранение и обработку. В отличие от текстового хранения JSON, двоичный JSONB индексируется и позволяет выполнять быстрые запросы по ключам внутри JSON-объектов. Для нашего приложения такой подход является важным, так как в работе нужно сохранять исходные JSON-документы в первоначальном виде и при этом выполнять по ним выборки и фильтрацию без ущерба производительности. Кроме того, JSONB автоматически убирает лишние пробелы и сортирует ключи, что экономит место и унифицирует данные. Возможность создать индекс GIN или B-tree по полям внутри JSONB обеспечивает быстрый доступ к данным даже при сложных структурах [27].

Во-вторых, надежность и соответствие ACID. PostgreSQL является полноценной реляционной СУБД, поддерживающей транзакции, целостность данных и сложные запросы на языке SQL. Поскольку в проекте планируется хранить историю вычислений и рекомендации модели, целостность этих данных важна. PostgreSQL гарантирует, что данные не потеряются и не разойдутся даже при одновременных обращениях к базе из разных частей приложения. В сравнении с NoSQL-решениями, PostgreSQL предлагает более строгую структуру и надежность транзакций, что для выпускной работы предпочтительно, ведь важнее корректность, чем абсолютная гибкость схемы [28].

В-третьих, расширяемость и поддержка сложных запросов. На SQL-сервере можно легко выполнять агрегатные и аналитические запросы. Например, собрать статистику по частоте возникновения коллизий у каждой хеш-функции. PostgreSQL поддерживает хранимые функции, представления, сложные запросы, соответственно такой набор инструментов может быть полезным, если анализ JSON-объектов расширится. К примеру, можно реализовать на стороне базы хранимую функцию, считающую хеши определённым алгоритмом для всех записей, и выбирать из неё результаты [29].

В-четвертых, собирая статистику, было выявлено, что в текущих реалиях данную СУБД используют во многих областях. PostgreSQL достаточно распространена, значит есть много информации в открытом виде. Это сократит время на настройку базы. Spring Boot имеет готовые стартовые модули для PostgreSQL, что означает минимум усилий для подключения [30]. Для работы достаточно указать URL, логин и пароль в конфигурации, далее, при запуске приложения, можно использовать JdbcTemplate или репозитории Spring Data JPA для взаимодействия с базой. К тому же PostgreSQL отлично работает в Docker-контейнерах, и для неё легко найти образы и утилиты бэкапов.

Наконец, нельзя не отметить, что PostgreSQL является бесплатной и открытой СУБД, что соответствует критериям дипломного проекта. Она кроссплатформенная, поэтому после разработки на определенной ОС на составит перенести и запустить приложение на другую. Разработка велась на Windows/Ubuntu, а развёртывание может быть и на Linux-сервере .

Таким образом, PostgreSQL была выбрана как оптимальная СУБД для дипломного проекта. Она сочетает преимущества реляционных баз с возможностями хранения JSON-документов (гибкость схемы), что идеально подходит под задачу хранения JSON-объектов и результатов их анализа. В дальнейшем, если объём данных сильно возрастет, PostgreSQL сможет

масштабироваться вертикально или горизонтально, однако для прототипа дипломного проекта это не потребуется.

2.6 Выбор инструментов для тестирования и разработки

Для успешной реализации проекта был выбран набор инструментов, облегчающих написание кода, его отладку и тестирование API. В их числе – интегрированные среды разработки (IDE) для разных языков, редакторы кода и специальные утилиты для тестирования веб-сервисов. Правильный выбор инструментов повышает продуктивность и позволяет сосредоточиться на логике приложения, поэтому ниже приводится обоснование, почему были выбраны именно эти средства.

IntelliJ IDEA – основная IDE, применявшаяся при разработке серверной части на Java. IntelliJ IDEA от компании JetBrains давно зарекомендовала себя как одна из самых мощных и удобных сред для Java-разработки. Она широко используется профессиональными разработчиками и студентами благодаря богатому функционалу и простоте использования [31]. В контексте нашего проекта IntelliJ предоставила такие возможности, как умное автодополнение кода, рефакторинг, подсветку ошибок, интеграцию с системой сборки Maven и встроенные средства для запуска и отладки Spring Boot-приложения. Огромным плюсом является поддержка плагинов, таких как, Spring Tools который автоматически подсказывает настройки конфигурационных файлов, а Database Tools в IntelliJ позволяет прямо из IDE подключаться к PostgreSQL и просматривать таблицы. Всё это существенно ускорит разработку бэкенда. Кроме того, интуитивно понятный интерфейс IntelliJ и её производительность даже на большом проекте сделали работу комфортной. Единственный минус, который есть в данном инструменте это то, что полная версия IntelliJ Ultimate является платной, но для студентов JetBrains предоставляет бесплатную лицензию. В итоге IntelliJ IDEA стала незаменимым инструментом, поддерживая высокий темп разработки Java-кода.

PyCharm – IDE от той же компании JetBrains, которая будет использована для разработки Python-модуля ИИ. Данная IDE является специализированной средой для Python, которая обеспечивает богатую функциональность, такие как автодополнение и статический анализ кода, мощный отладчик, интеграцию с системами контроля версий и поддержку популярных веб-фреймворков. Значимым является то, что PyCharm изначально поддерживает технологии веб-разработки на Python, включая Flask. Также данный выбор упрощает работу с такими библиотеками, как Pandas и scikit-learn, благодаря специальному режиму Scientific Mode, где можно исполнять фрагменты кода и сразу видеть графики или вывод в интерактивной консоли. Как и IntelliJ, PyCharm обладает всеми возможностями для рефакторинга и удобного перехода по коду, что помогает поддерживать чистоту проекта. Важным моментом PyCharm является то, что он помогает подсвечивать потенциальные ошибки, неявные приведения типов или несовместимость версий библиотек. Это предотвращает многие проблемы еще до запуска программы. Стоит отметить, что для студентов JetBrains также предоставляет PyCharm Professional бесплатно, поэтому в рамках дипломной работы будет полный доступ ко всем функциям IDE.

Помимо «тяжелых» IDE, в проекте применялся и более легковесный редактор кода Visual Studio Code (VS Code). Он был использован для разработки фронтенда на React и редактировании сопутствующих файлов. VS Code популярен благодаря своей скорости, гибкости и бесплатности. В данной IDE есть поддержка плагинов для Docker. С помощью этих плагинов можно настроить VS Code именно под нужды дипломной, превратив его в удобный инструмент для фронтенда. На этапе разработки интерфейса React приложение запускалось с помощью утилиты `npm start`, и VS Code отлично подойдет для редактирования компонентов, быстрого просмотра результатов в браузере и отладки через встроенные DevTools. Отдельно стоит подчеркнуть невысокие требования VS Code к ресурсам системы, в технических характеристиках минимальные требования совсем не большие

[32]. Это позволит параллельно держать запущенными IntelliJ и PyCharm для бэкенда, и VS Code для фронтенда. Таким образом, VS Code стал легким, но эффективным дополнением к основным IDE, особенно для задач фронтенда.

Для тестирования REST API, созданного в приложении, применялся инструмент Postman. Postman – это популярная программа с графическим интерфейсом, предназначенная для отправки HTTP-запросов и проверки ответов от сервера. Она широко используется как разработчиками, так и тестировщиками при работе с веб-сервисами. С его помощью можно будет вручную эмулировать запросы от фронтенда к бэкенду, Тестировать отправки GET и POST запросов Spring Boot и Flask-сервиса, передавать в теле JSON-объекты, заголовки, и сразу видеть, что возвращает сервер. Удобство Postman в том, что все запросы можно сохранять в коллекции и повторять многократно, меняя параметры. Для дипломного проекта будет создана коллекция запросов для основного функционала. Каждый такой запрос можно будет протестировать отдельно, удостоверившись, что API работает корректно. Postman позволяет не только видеть ответ (включая тело, статус, заголовки), но и писать простые тест-скрипты на JavaScript, чтобы автоматизировать проверку (например, убедиться, что в ответе всегда статус 200 и определенные поля присутствуют). В целом, данный инструмент значительно упростит отладку. Кроме перечисленных инструментов, будет применяться технология, которая нужна для поддержки и запуска кода. Система контроля версий Git нужна для сохранения истории изменений кода. Визуальное управление базой данных PostgreSQL на этапе разработки будет осуществляться через программу pgAdmin 4. Для отладок сложных ситуаций при HTTP-взаимодействии Yandex браузер предоставляет инструменты разработчика, которые наглядно показывают весь процесс взаимодействия двух частей приложения.

Все вышеперечисленные инструменты в совокупности создадут комфортную среду разработки. Для дипломной работы важно не только

решить поставленную задачу, но и показать владение профессиональным инструментарием, такими как IntelliJ, PyCharm, VS Code, Postman и др.

2.7 Средства развертывания приложения

Для развертывания приложения бы сделан выбор в пользу Docker. Docker – это платформа контейнеризации, позволяющая упаковать приложение со всеми его зависимостями в стандартный контейнер, который можно запустить на любой машине. По сути, Docker создает легковесные виртуализированные окружения для каждого сервиса, избавляя от проблем совместимости. Для дипломного проекта Docker идеально подошел, так как система состоит из нескольких компонент, написанных на разных языках и требующих разных сред исполнения. Будет созданы Docker-образы для:

- Java сервиса,
- Python сервиса,
- базы данных.

Затем с помощью Docker Compose эти образы будут определяться как сервисы единого приложения, чтобы их можно было запускать одной командой, и они автоматически видели друг друга в одной сети. Таким образом, Docker даст возможность развернуть полное окружение для работы приложения локально Docker имеет ряд преимуществ по сравнению с традиционными подходами вроде XAMPP [33].

Во-первых, контейнеры обеспечивают чистую изоляцию сред, так как каждый сервис работает в своём контейнере, со своим набором библиотек. Это исключает конфликты версий [34].

Во-вторых, Docker хорошо масштабируется и подходит не только для разработки, но и для дальнейшего запуска приложения для пользователей. Docker значительно облегчает развертывание на сервере, если в будущем приложение будет выкладываться на удаленный сервер или в облако, его можно будет практически без изменений запустить на них.

Таким образом для запуска приложения Docker будет полностью подходить как для целей дипломной работы, так и для дальнейшей модернизации приложения.

3 Структура системы

Разрабатываемая подсистема представляет собой веб-приложение, использующее методы искусственного интеллекта (ИИ) для анализа JSON-объектов и автоматического выбора оптимальной хеш-функции. В контексте данной системы задача ИИ состоит в том, чтобы по характеристикам JSON-данных подобрать наиболее подходящий алгоритм хеширования – то есть такой, который даст наименее коллизийный и наиболее эффективный результат для этих данных.

Подсистема спроектирована без разделения на роли пользователей, то есть все взаимодействие осуществляется от лица единственного типа акторов – конечного пользователя. Пользователь взаимодействует с системой через удобный веб-интерфейс (в браузере) либо напрямую посредством REST API запросов. В данной главе рассматривается структура подсистемы, где описаны основные функциональные возможности, компоненты системы и их взаимодействие. Также приведены диаграммы прецедентов и компонентов, иллюстрирующие работу системы, и описана последовательность действий пользователя при использовании приложения.

3.1 Основные функции подсистемы

Система предоставляет пользователю ряд ключевых функций, позволяющих загрузить данные, получить рекомендации по выбору хеш-алгоритма, выполнить хеширование и проанализировать результаты. Ниже перечислены основные функциональные возможности системы:

- загрузка JSON-объекта. Пользователь может загрузить JSON-объект через веб-интерфейс. Например, это может быть файл с данными в формате JSON или JSON-строка, вставленная в специальное поле. Система принимает JSON-объект и валидирует его структуру. После успешной загрузки данные готовы для дальнейшего анализа,

- запрос подбора подходящей хеш-функции. Пользователь отправляет запрос на автоматический подбор оптимальной хеш-функции для загруженного JSON. На сервере вызывается модель ИИ, которая анализирует

структуру и содержимое JSON-объекта. На основе заложенных в модель критериев, она генерирует рекомендацию. Им является список наиболее подходящих алгоритмов хеширования под данные пользователя,

- получение рекомендаций и выбор алгоритма. Система возвращает пользователю сгенерированные рекомендации в виде списка алгоритмов хеширования, отсортированного по степени пригодности. Например, в списке могут присутствовать ранее рассмотренные алгоритмы, снабжённые краткими характеристиками и оценками. Пользователь изучает предложенные варианты и выбирает из списка один алгоритм, который считает оптимальным,

- выполнение хеширования выбранным методом и получение результата. После выбора алгоритма пользователь может запустить процедуру хеширования. Система посредством выбранной хеш-функции вычисляет хеш-код для исходного JSON-объекта. В результате пользователь получает на экране сгенерированный хеш-код. Дополнительно могут отображаться сопутствующие показатели, например, время выполнения хеширования, длина полученного хеш-кода, и т. п., что позволяет оценить эффективность метода,

- сравнение поведения разных хеш-функций. Подсистема предоставляет возможность визуального сравнения нескольких хеш-функций на одном и том же наборе данных JSON. Пользователь может выбрать опцию сравнения – например, отметить галочками несколько алгоритмов из списка рекомендаций для параллельного анализа. Система тогда выполнит хеширование JSON-данных каждым из выбранных алгоритмов, собирая статистику (время выполнения, количество коллизий при условной хеш-таблице фиксированного размера, равномерность распределения хешей и др.). Результаты сравнения наглядно представляются пользователю в виде таблиц и графиков. Например, может строиться график, показывающий распределение значений хеш-функции, или диаграмма, сравнивающая время выполнения разных алгоритмов на данных пользователя. Это позволяет

наглядно увидеть, какой из алгоритмов работает быстрее или даёт более равномерное распределение хеш-значений,

– сохранение или экспорт результатов. Пользователь может сохранить результаты работы системы для последующего использования. Предусмотрены функции экспорта: например, пользователь может скачать отчет в виде файла (JSON, CSV либо PDF отчёт) с указанием выбранного алгоритма, полученного хеш-кода, метрик работы (время, число коллизий и т. д.) и прочей информации. Также возможно сохранение результатов сравнения. Эта функция упрощает интеграцию результатов работы подсистемы в другие приложения или включение полученных данных в отчёты пользователя.

На рисунке 8 показана упрощенная диаграмма прецедентов.

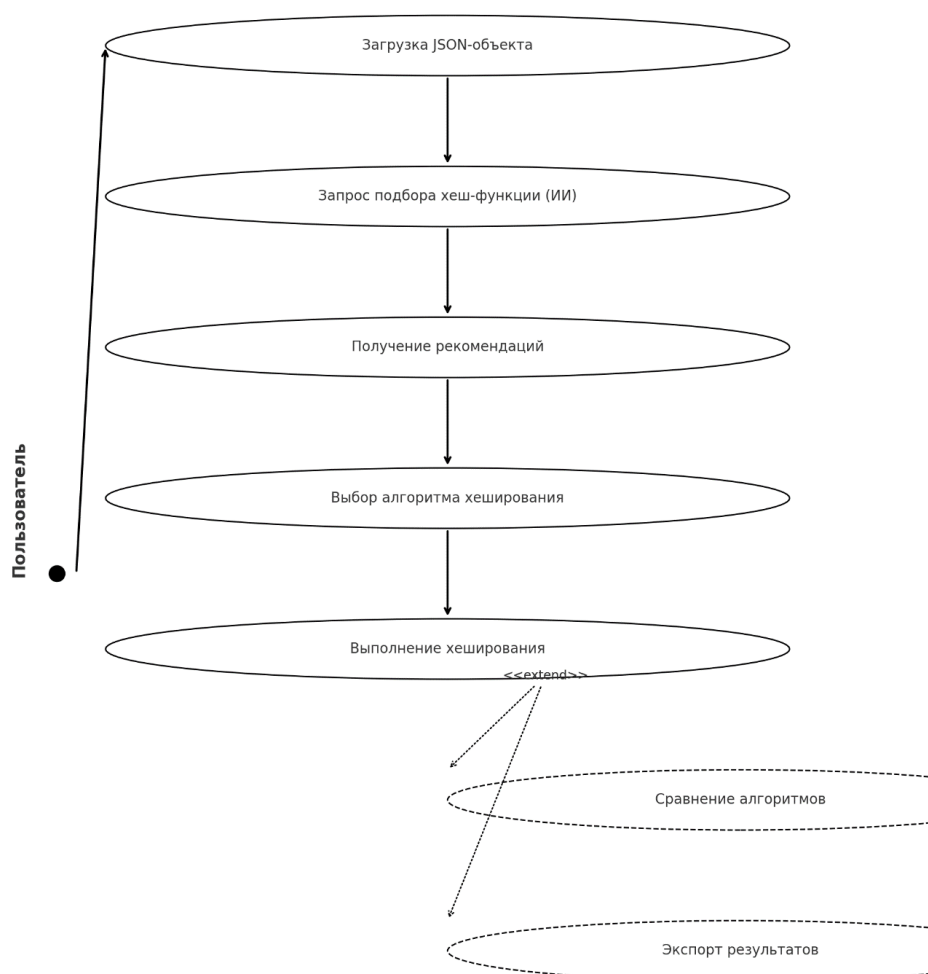


Рисунок 8 – Упрощенная диаграмма прецедентов

Нужно отметить, что описанный сценарий работы не требует регистрации или авторизации – система сразу готова к работе. Взаимодействие может циклически повторяться, пользователь может поэкспериментировать с разными JSON-файлами и алгоритмами, выполняя описанные шаги сколько угодно раз.

3.2 Архитектура и компоненты системы

Внутренняя структура подсистемы реализована по принципу клиент-сервер. Клиентская часть – это веб-интерфейс, с которым взаимодействует пользователь через браузер. Серверная часть предоставляет REST API, обрабатывающее запросы от клиента, и выполняет основную логику. Компоненты системы представлены на рисунке 9.

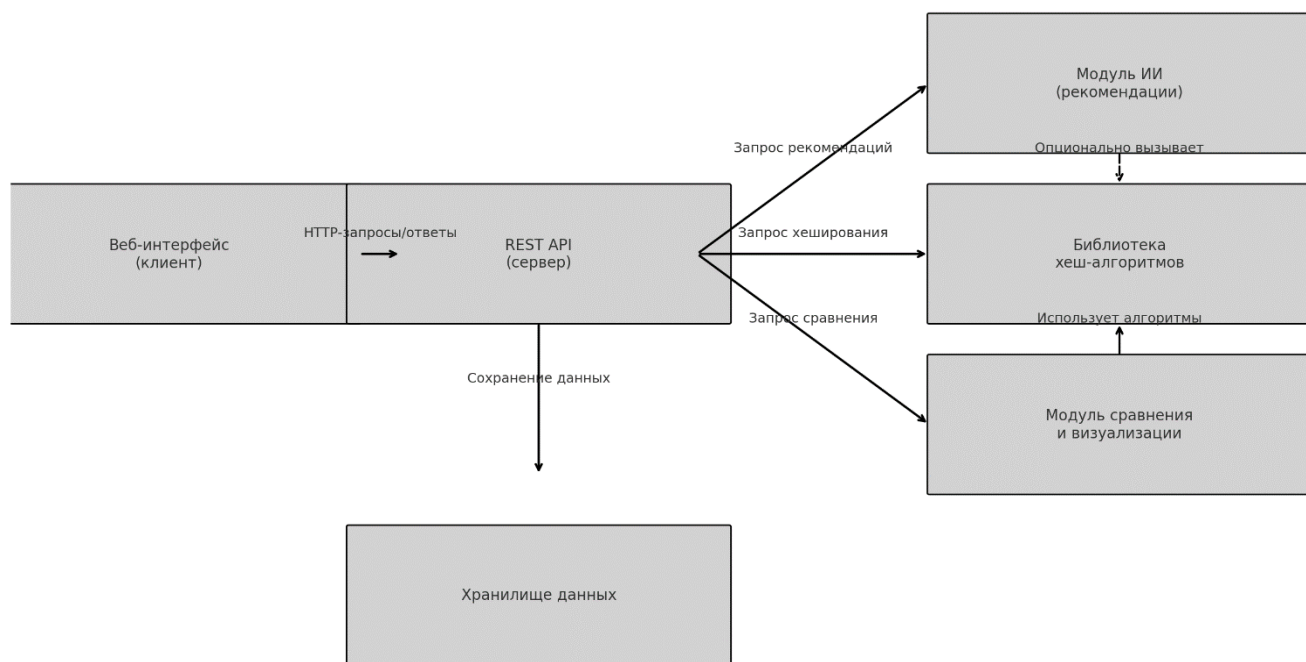


Рисунок 9 – Диаграмма компонентов системы

Ниже описаны основные компоненты системы и их взаимосвязи:

- клиентская часть: предоставляет пользовательский интерфейс для работы с системой. Реализован в виде одностраничного веб-приложения или набора веб-страниц, где доступны все функции: форма для загрузки JSON, кнопка для запуска подбора хеш-функции, интерфейс показа списка рекомендаций, элементы управления для выбора алгоритма, кнопка для выполнения хеширования, а также модули визуализации для сравнения

алгоритмов. Веб-интерфейс отвечает за удобство взаимодействия с пользователем. Он отправляет соответствующие запросы на сервер и отображает полученные ответы. Например, после загрузки JSON-объекта интерфейс может сразу отобразить структуру данных или размер файла, после получения списка рекомендуемых алгоритмов показать их в виде выпадающего списка или карточек с описанием. После хеширования – показать полученный хеш-код и статистику для сравнения. Важной частью клиентской логики являются средства визуализации, так как с их помощью результаты сравнения алгоритмов представляются понятно (например, с помощью библиотек JavaScript для построения графиков),

– серверная часть: предоставляет набор HTTP-путей для выполнения всех основных операций. По сути, это сервер приложений, принимающий запросы от веб-интерфейса и возвращающий результаты в формате JSON. Основные пути могут включать в себя загрузку JSON (POST /upload-json), запуск анализа и получения рекомендаций (GET /recommendations), выполнение хеширования выбранным алгоритмом (POST /hash), получение данных для сравнения алгоритмов (GET /compare), и экспорт результатов (GET /export). Сервер валидирует входящие данные, затем вызывает необходимые внутренние компоненты и формирует ответ. Таким образом, REST API является связующим звеном между клиентом и внутренней логикой системы,

– модуль ИИ – отвечает за анализ JSON-объекта и выбор подходящих хеш-функций. Это компонент, встроенный в серверное приложение. В простейшем случае может быть реализован на основе правил. В более сложном варианте модуль ИИ использует модель машинного обучения, обученную на множестве примеров, где каждому типу данных сопоставлены наиболее эффективные хеш-функции. При вызове этот модуль получает на вход, загруженный JSON и вычисляет набор характеристик, такие как размер данных, типы и распределение значений, энтропию строк и т. д. На основе этих характеристик модель выдает рейтинг алгоритмов

хеширования. В результате модуль возвращает список рекомендаций – например, несколько названий алгоритмов с указанием рейтинга пригодности. Этот список затем передается через API клиенту,

- библиотека хеш-алгоритмов – компонент серверной части, содержащий реализации различных хеш-функций, доступных в системе. Здесь собраны все алгоритмы, из которых осуществляется подбор как криптографических, так и некриптографических, оптимизированные для скорости и низкой коллизии. Библиотека предоставляет унифицированный интерфейс для вычисления хешей, например, другие компоненты могут вызывать функцию вычисления хеша, передавая на вход данные JSON и выбирая алгоритм по названию или идентификатору. Таким образом, добавление нового алгоритма хеширования в систему сводится к его реализации и регистрации в данной библиотеке. В рамках сравнения алгоритмов библиотека также может предоставлять функции для получения статистики,

- хранилище данных: система хранения, используемая подсистемой. Данный компонент может быть реализован в различных формах: простое хранение в памяти, файловое хранилище либо база данных. Поскольку система не управляет множеством пользователей и не требует долгосрочного сохранения большого объёма данных, возможно, достаточно хранить информацию в оперативной памяти сервера в течение сеанса. Например, загруженный JSON-объект может сохраняться в памяти и привязываться к уникальному ID сессии, чтобы последующие запросы использовали эти же данные. Результаты хеширования и сравнения также могут временно храниться, особенно если пользователь планирует их экспортировать. В случае реализации функции экспорта через скачивание файлов, хранилище может содержать сгенерированные отчёты или изображения графиков до момента передачи их пользователю. Если же необходимо более длительное хранение, может применяться простая база данных или запись на диск. В любом случае, данный компонент

обеспечивает сохранность данных между этапами обработки в рамках одной сессии пользователя,

Таким образом, спроектированная структура подсистемы обеспечивает пользователю возможность легко загрузить свои JSON-данные, получить обоснованную рекомендацию по выбору хеш-функции, выполнить хеширование и проанализировать эффективность различных алгоритмов.

4 Разработка программного обеспечения

При разработке программного обеспечения было создано два сервиса. Первый был разработан на язык Java, в котором находилось веб-часть для представления и обработки данных. Во втором была реализована работа искусственного интеллекта.

4.1 Разработка веб-сервиса

Разработанный Java-сервис предназначен для анализа JSON-объектов с помощью методов искусственного интеллекта и генерации для них хеш-значений. Главная цель сервиса – рекомендовать оптимальный алгоритм хеширования для заданного JSON и выдавать специально сгенерированный хеш (custom hash) этого объекта. Данный сервис играет ключевую роль в общей системе, отвечая за интеграцию интеллектуальной модели в процесс хеширования данных. Например, в типичной информационной системе сервис может обеспечивать контроль целостности данных, например вместо использования фиксированного алгоритма хеширования, он динамически подбирает подходящий метод на основе анализа структуры и содержимого JSON. Это позволяет потенциально снизить количество коллизий и улучшить равномерность распределения хешей для разнообразных входных данных.

Приложение реализовано с использованием фреймворка Spring (Spring Boot) и следует многослойной архитектуре с чётким разделением ответственности между слоями. Выделены следующие основные уровни:

- уровень представления: обеспечивает прием входящих HTTP-запросов и формирование HTTP-ответов. В нашем сервисе эту роль выполняют REST-контроллеры, обрабатывающие вызовы API. Они выполняют начальную обработку (например, валидацию) и передают данные на следующий уровень логики,
- уровень бизнес-логики: содержит основную логику приложения, не завязанную на детали веб-протокола. Сервисы реализуют операции над данными: вызывают внешние API, выполняют вычисления, вызывают

методы хранения и т.д. В данной системе к сервисам относятся компоненты, которые обращаются к модели ИИ и рассчитывают метрики хеширования,

- уровень моделей данных: включает классы моделей и DTO (Data Transfer Object), с помощью которых данные передаются между слоями (например, структуры запросов и ответов в формате JSON). Эти классы, как правило, не содержат логики, а только поля и методы доступа к ним.

Приложение организовано по принципу MVC (Model-View-Controller). View представлен как клиентская часть представлена отдельной веб-страницей. Контроллеры (HashController, AIMetricsController) помечены аннотацией `@RestController` и получают HTTP-запросы на определенные URL. Они используют сервисы (RecommendationService, AIMetricsService), помеченные аннотацией `@Service`, для выполнения необходимой работы. Связь между контроллерами и сервисами реализована через внедрение зависимостей Dependency Injection. На рисунке 10 представлена диаграмма компонентов, отражающая архитектуру системы в целом. На ней показано, как различные части взаимодействуют друг с другом.

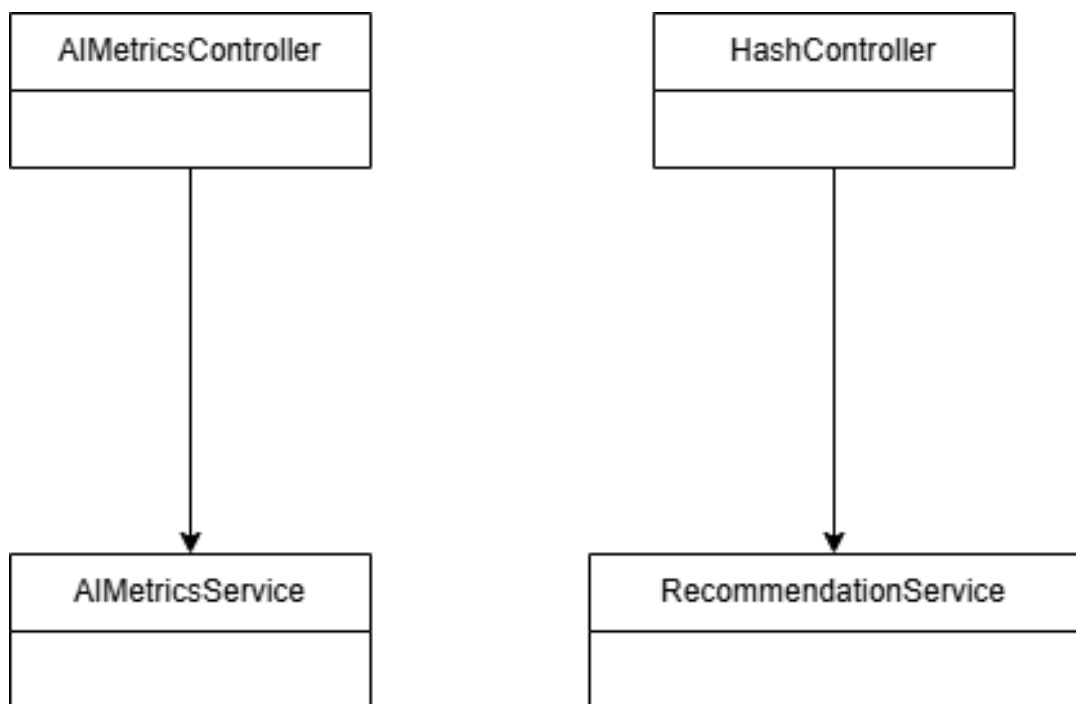


Рисунок 10 – Диаграмма компонентов

Сервис предоставляет ряд REST API конечных точек (endpoints), с помощью которых внешние клиенты (например, фронтенд или другие

сервисы) могут воспользоваться функциональностью рекомендаций и анализа хешей. Ниже перечислены основные URL и их предназначение:

- POST /api/hash/recommend – принимает на вход JSON-объект и возвращает рекомендацию по алгоритму хеширования вместе с вычисленным custom hash. Приводя пример, клиент отправляет некоторый JSON, а сервис отвечает, например: «Рекомендуемый алгоритм: SHA-256, Custom Hash: XZY..., Комментарий: ...». В ответе содержится название алгоритма, который модель сочла оптимальным для этого JSON, сгенерированное хеш-значение (обычно с использованием кастомного алгоритма) и дополнительный комментарий,

- GET /ai-metrics/benchmark/{n} – выполняет бенчмарк-тестирование алгоритмов хеширования на n случайно сгенерированных JSON-объектах и возвращает собранные метрики. Клиент указывает число n (например, 10000), после чего сервис генерирует (через модель ИИ) указанное количество различных JSON-структур, хеширует их разными алгоритмами и подсчитывает статистику: сколько уникальных хешей получилось у каждого алгоритма, сколько коллизий произошло, а также среднее время вычисления хеша. Результат возвращается в виде JSON-объекта, где для каждого алгоритма приведены эти показатели.

Вся коммуникация в API следует архитектурному стилю REST, то есть используются понятные URL, стандартные методы, такие как POST для отправки данных и вычисления результатов и GET для получения информации, а данные передаются в формате JSON. Это обеспечивает совместимость с любыми HTTP-клиентами и простоту интеграции сервиса в более широкую систему.

Основная идея сервиса заключается в том, чтобы с помощью методов искусственного интеллекта улучшить выбор или разработку алгоритма хеширования под конкретные данные. Модель ИИ анализирует структуру и содержимое входного JSON и выдает два результата: рекомендуемый алгоритм и собственно хеш-значение. Рекомендуемый алгоритм

представляет собой название наиболее подходящего стандартного алгоритма хеширования, который модель предлагает использовать для данного объекта. Модель была обучена распознавать, какие типы данных лучше обрабатывать определённой функцией, а для каких достаточно более простого алгоритма. Вторым результатом представляет собой хеш, полученный с помощью обученного искусственного интеллекта, разработанного в рамках данной работы.

Взаимодействие с моделью искусственного интеллекта происходит в строго определённых местах приложения. Для отправления запросов на сервис с ИИ используется внутренний класс `RestTemplate`. Таким образом, ИИ-модель встроена в контур работы сервиса как удалённый компонент, отвечающий за интеллектуальные решения. Наш Java-сервис фактически делегирует модели ИИ задачи. Это позволяет изолировать сложные вычисления и использовать, к примеру, Python-библиотеки для машинного обучения, не переписывая их на Java. Код сервиса представлен в приложении А.

Для удобства демонстрации работы сервиса был разработан простой визуальный интерфейс – веб-страница, через которую пользователь может в интерактивном режиме воспользоваться функциональностью API. Эта страница содержит HTML-разметку, CSS-стили для оформления и сценарии на JavaScript для вызова REST API. Интерфейс состоит из двух основных разделов, соответствующих возможностям сервиса:

- рекомендация и генерация хеша по JSON,
- сравнительный бенчмарк алгоритмов.

В первой секции страницы пользователь видит заголовок и многострочное текстовое поле, куда можно вставить или ввести JSON-объект. Во второй секции страницы пользователь может протестировать эффективность алгоритмов на случайных данных.

4.2 Разработка клиентской части

Для приложения был создан интерфейс, который помогает взаимодействовать пользователю с логикой программного обеспечения, на рисунке 11 представлен пример.

Оптимизация алгоритмов хеширования и сравнительный анализ

1. Рекомендация и генерация хеша по вашему JSON

Вставьте JSON:

```
{ "name": "Test", "value": 123 }
```

Получить рекомендации

Рекомендованный алгоритм: Object.hashCode
Custom Hash: 52df48133bc5ba42767b86898484838a
Комментарий: ИИ выбрал лучший алгоритм и сгенерировал собственный хеш-код

2. Сравнительный бенчмарк алгоритмов

Сколько JSON сгенерировать для теста:

Запустить бенчмарк

Алгоритм	Уникальных	Коллизий	Время (мкс)
custom_keras	9999	0	10.2
custom_onnx	9999	0	0.9
sha256	9999	0	2.1
crc32	9999	0	0.1
murmur3	9999	0	0.3
fnv1a	9999	0	39.5
objectHashCode	9999	0	44.6

Рисунок 11 – Интерфейс

На рисунке представлены компоненты интерфейса:

- поле для ввода Json,
- кнопка для получения рекомендаций и сгенерированного ИИ хеш-кода,
- вывод работы рекомендаций,
- количество JSON, на которых будет проходить сравнение,
- запуск сравнения,
- результат сравнения.

Таким образом был реализован красивый вывод информации для основного функционала приложения. В целом, визуальный интерфейс получился интуитивно понятным, как студент, так и разработчик, испытывающий систему, может быстро проверить работу модели ИИ, вводя разные JSON и наблюдая, как меняется рекомендуемый алгоритм и хеш, и оценить качество алгоритмов, запуская сравнение на разном количестве данных. Это подчёркивает ориентацию работы на практическую реализацию и удобство, то есть не только описание алгоритмов, но и создание полноценного приложения, готового к взаимодействию с пользователем. Код клиентской части представлен в приложении А.

4.3 Разработка сервиса искусственного интеллекта

ИИ-модуль предназначен для увеличения качества и индивидуальности хеширования JSON-объектов. Он автоматически выбирает оптимальный классический хеш-алгоритм или генерирует собственный хеш, наиболее подходящий под конкретный формат данных. Это позволяет снизить число коллизий и учесть специфические особенности передаваемых JSON-структур. В модуле реализованы две модели: классификатор для выбора лучшего из стандартных алгоритмов и генератор уникального хеша по данным объекта. Такой подход обеспечивает более гибкое хеширование, чем статический алгоритм, и позволяет сервису адаптироваться к различным типам входных данных.

В таблице 2 приведена таблица основных сценариев Python-скриптов подготовки и обучения.

Таблица 2 – Описание скриптов

Скрипт	Назначение	Выходной формат
dataset_generator.py	Генерация синтетических JSON-примеров	JSON-файл
dataset_generator_real.py	Сбор и синтез реальных JSON-данных	JSON-файл
train_classifier.py	Обучение модели-классификатора выбора алгоритма	Обученная модель
train_and_export_to_onnx.py	Экспорт обученного классификатора в ONNX	Файл модели ONNX
train_custom_hash.py	Обучение модели-генератора собственного хеша	Обученная модель
inference_onnx.py	Скрипт инференса: загрузка ONNX-моделей и предсказание	Вывод хешей и меток

Шаги подготовки данных интуитивно понятны. Скрипты генерации формируют тренировочные примеры, а скрипты обучения превращают их в обученные модели. Использование синтетических данных оправдано их «ненасыщенностью» в некоторых задачах: как отмечается в литературе, синтетические данные искусственно восполняют недостаток реального набора, делая обученные ИИ-модели более устойчивыми.

Обучение ИИ-модуля разделено на две параллельные задачи:

- классификатор алгоритмов: Скрипт train_classifier.py обучает нейросеть определять, какой из существующих классических хеш-функционалов лучше всего применить к данному JSON. После обучения

скрипт `train_and_export_to_onnx.py` экспортирует модель в формат ONNX. Классификатор решает задачу многоклассовой классификации (каждому классу соответствует свой алгоритм), используя имеющиеся примеры, где на входе – особенности JSON (например, частота встречаемости значений в полях, длина текста и т. д.), а на выходе – метка лучшего алгоритма,

– генератор `custom`-хеша: Скрипт `train_custom_hash.py` обучает модель генерировать собственный хеш из содержимого JSON-объекта. Модель получает JSON-данные на входе и обучается выдавать уникальный код (например, фиксированной длины строку), отражающий сгенерированный хеш. Эта модель действует как «черный ящик», который стремится выдавать различные хеши для различных входов, добавляя «индивидуальность» по сравнению со стандартными алгоритмами. Обе модели по окончании обучения сохраняются в формате ONNX для последующего использования.

Процесс обучения логичен: сначала подготавливаются примеры (включая метки или контрольный хеш), затем модели обучаются на них. При этом классификатор учится «распознавать» структуры данных и ставит метку алгоритма, а генератор – создавать собственный хеш. В силу универсальной теоремы приближения нейросети могут, теоретически, моделировать любую функцию, однако в практических целях мы ограничиваемся конкретными связными архитектурами и выбором формата вывода.

Входные данные представлены в виде JSON-объектов с фиксированным набором полей. Например, каждый объект может содержать ключи вида `payload` (основной текст/данные), `metadata` (служебная информация) и т. д. Для классификатора в качестве меток используется имя алгоритма хеширования, а для генератора используется эталонный хеш. Тренировочный JSON-файл состоит из записей с парами, где метка – требуемый результат. Такой формат позволяет легко загружать данные в TensorFlow/PyTorch-пайплайн и подавать их на обучение.

Обе модели имеют стандартную нейросетевую архитектуру, достаточную для решения своих задач. Классификатор обычно состоит из слоев преобразования входных данных в признаки и нескольких полносвязных слоев. Так, после энкодирования JSON (например, через текстовое эмбединг-представление) идут скрытые слои (Dense + ReLU), а на выходе – слой Softmax с числом нейронов, равным количеству алгоритмов. Такая архитектура позволяет вычислить вероятности каждого алгоритма и выбрать наиболее подходящий.

Генератор custom-хеша часто строится по схеме «encoder–decoder»: входной JSON проходит через рекуррентные слои (LSTM/GRU), после чего декодер выдает последовательность символов хеша или фиксированный вектор, который интерпретируется как хеш. К примеру, можно использовать несколько LSTM-слоев и на выходе линейный слой, генерирующий вектор заданной длины (который далее конвертируется в шестнадцатеричную строку или иные символы). Формат вывода настраивается так, чтобы получался уникальный (или максимально разнообразный) хеш для разных входов. При этом детали, такие как число слоев и размер слоев, подбираются опытным путем, но они не играют ключевой роли в описании: важнее тип слоев (рекуррентные и полносвязные) и общее назначение архитектуры.

Обе модели оптимизируются с помощью методов обратного распространения ошибки: классификатор – по функции кросс-энтропии, генератор – по функции, которая оценивает различие между сгенерированным хешем и эталонным. Гиперпараметры подбираются отдельно, но в описании их можно не приводить, поскольку они зависят от эксперимента.

После обучения обе модели экспортируются в универсальный формат ONNX. Во время инференса служебный скрипт `inference_onnx.py` загружает ONNX-модель и выполняет предсказания на новых JSON-данных: сначала классификатор определяет лучший алгоритм, затем, при необходимости, генерируется кастомный хеш. ONNX-модели легко интегрируются в Java-

сервис через ONNX Runtime. ONNX Runtime предоставляет Java-API для запуска инференса ONNX-моделей на JVM, что позволяет напрямую вызывать обученную модель из кода сервера. В Java-приложении создается `OrtEnvironment`, затем загружается модель (`OrtSession`), и с помощью метода `run()` подаются входные тензоры (предварительно сформированные из JSON). Результатом являются выходные тензоры с метками или вектором хеша, которые интерпретируются и возвращаются клиенту.

Таким образом, на этапе инференса Java-сервис при поступлении запроса превращает JSON в нужный формат тензора, передает его в ONNX-модель и получает ответ. Этот процесс прозрачно работает благодаря ONNX-формату и библиотеке ONNX Runtime.

Использование формата ONNX обусловлено его стандартностью и переносимостью. ONNX – это открытый формат для представления моделей машинного обучения, который определяет общее множество операторов и файл модели. Благодаря этому модели, обученные в одной среде (PyTorch, TensorFlow и т. д.), можно запускать в другой без дообучения или портирования. ONNX обеспечивает интероперабельность: разработчик может не заботиться о том, на каком фреймворке модель обучалась, и использовать соответствующий движок инференса. Код сервиса представлен в приложении Б.

Кроме того, ONNX обладает поддержкой оптимизаций и исполнителей на разных платформах: для нашего случая важна именно интеграция с Java. ONNX Runtime имеет Java-биндинг, позволяющий эффективно выполнять инференс на JVM. Это избавляет от необходимости самостоятельно переписывать модель под Java или запускать отдельный Python-процесс.

5 Тестирование

Для оценки эффективности реализованной системы был проведён бенчмарк различных алгоритмов хеширования с использованием заранее сгенерированных JSON-объектов, имитирующих реальные данные различной структуры и сложности. Генерация осуществлялась случайным образом с варьированием глубины вложенности, числа ключей и типов значений (строки, числа, логические значения, массивы, вложенные объекты), что обеспечивало разнообразие тестовых случаев. Основной целью тестирования являлось комплексное сравнение алгоритмов по ключевым характеристикам:

- количество уникальных значений хеша (важно для равномерности и устойчивости к коллизиям),
- число коллизий (сколько раз разные JSON-объекты получили одинаковый хеш),
- скорость работы (в микросекундах).

В сравнении участвовали следующие алгоритмы:

- `Object.hashCode`: встроенный метод Java для генерации хеша,
- SHA-256: криптографический алгоритм хеширования,
- CRC32: простой алгоритм контрольной суммы,
- FNV-1a, Murmur3: популярные быстрые хеши,
- Custom Hash (ONNX): модель нейросети, экспортированная в ONNX-формат,
- Custom Hash (Keras): оригинальная Python-версия кастомного хеш-функционала.

На рисунке 13 представлен тест для 1000 сгенерированных JSON.

Оптимизация алгоритмов хеширования и сравнительный анализ

1. Рекомендация и генерация хеша по вашему JSON

Вставьте JSON:

```
{ "name": "Test", "value": 123 }
```

Получить рекомендации

Рекомендованный алгоритм: Object.hashCode

Custom Hash: 52df48133bc5ba42767b86898484838a

Комментарий: ИИ выбрал лучший алгоритм и сгенерировал собственный хеш-код

2. Сравнительный бенчмарк алгоритмов

Сколько JSON сгенерировать для теста:

Запустить бенчмарк

Алгоритм	Уникальных	Коллизий	Время (мкс)
custom_keras	1000	0	79.4
custom_onnx	1000	0	1.9
sha256	1000	0	2.8
crc32	1000	0	0.6
murmur3	1000	0	0.3
fnv1a	1000	0	43.7
objectHashCode	1000	0	47.5

Рисунок 13 – Тест для 1000 JSON

В самом первом тесте было сгенерировано 1000 случайных JSON-объектов. Практически все алгоритмы показали идеальный результат:

- все 7 алгоритмов (включая `custom_keras` и `custom_onnx`) выдали по 1000 уникальных хешей,
- коллизий не было вообще,
- быстрее всех оказался `custom_onnx` (1.9 мкс), на втором месте — `sha256` и `crc32` (~2.0-2.8 мкс),
- медленнее всех — `custom_keras` (79.4 мкс) и `Object.hashCode` (47.5 мкс).

Вывод: на малом объёме данных все алгоритмы обрабатывают отлично. Но уже видно, что нейросетевая версия ONNX работает значительно быстрее Keras. Это важно, если речь идёт об интеграции в реальные сервисы.

На рисунке 14 представлен тест для 10000 JSON.



2. Сравнительный бенчмарк алгоритмов

Сколько JSON сгенерировать для теста:

Запустить бенчмарк

Алгоритм	Уникальных	Коллизий	Время (мкс)
<code>custom_keras</code>	10000	0	10.1
<code>custom_onnx</code>	10000	0	0.9
<code>sha256</code>	10000	0	2.2
<code>crc32</code>	10000	0	0.7
<code>murmur3</code>	10000	0	0.3
<code>fnv1a</code>	10000	0	40.1
<code>objectHashCode</code>	9998	2	44.8

Рисунок 14 – Тест для 10000 JSON

При увеличении объёма в 10 раз уже начинают появляться первые различия:

- алгоритмы `custom_keras`, `custom_onnx`, `sha256`, `crc32`, `murmur3` и `fnv1a` – по-прежнему с нулём коллизий,
- `Object.hashCode` – первые 2 коллизии.
- по скорости всё остаётся стабильно: `onnx` – 0.9 мкс, `keras` – 10.1, `Object.hashCode` – 44.8 мкс.

Вывод: при переходе к 10 тысячам записей начинают проявляться слабые места стандартного Java-алгоритма, в то время как нейросети и криптография по-прежнему демонстрируют устойчивость. Кроме того, Keras снова показывает стабильную, но более медленную работу.

На рисунке 15 представлен тест для 100000 JSON.



2. Сравнительный бенчмарк алгоритмов

Сколько JSON сгенерировать для теста:

Запустить бенчмарк

Алгоритм	Уникальных	Коллизий	Время (мкс)
<code>custom_keras</code>	99974	0	3.4
<code>custom_onnx</code>	99974	0	1.0
<code>sha256</code>	99974	0	2.2
<code>crc32</code>	99973	1	0.7
<code>murmur3</code>	99972	2	0.4
<code>fnv1a</code>	99973	1	40.2
<code>objectHashCode</code>	99948	26	45.6

Рисунок 15 – Тест для 100000 JSON

Этот объём уже ближе к промышленному масштабу:

- objectHashCode дал 52 коллизии – и отстаёт от остальных,
- custom_keras и custom_onnx выдали 0 коллизий при 99 974 уникальных,
- время: custom_onnx – всё те же 1.0 мкс, sha256 — 2.2 мкс, Object.hashCode – 45.6 мкс.

Вывод: становится очевидно, что стандартный Object.hashCode не подходит для серьёзной нагрузки. ИИ-модель показывает, что даже при увеличении объёма она держит результат без потерь в уникальности.

На рисунке 16 представлен последний тест для 2000000 JSON.



Рисунок 16 – Тест для 2000000 JSON

Финальный тест показывает максимальную нагрузку. При таком объеме происходят критические расхождения:

- Object.hashCode снова провалился: более 4000 коллизий – это уже неприемлемо,
- crc32, fnv1a, murmur3 – в среднем 400-500 коллизий,
- только sha256 и кастомные модели (keras/onnx) – почти идеальные, с 0 или 21 коллизией.

Финальный вывод:

- Object.hashCode годится только для простых задач, где объёмы небольшие,
- sha256 хорош, но требует больше ресурсов,
- Custom_keras и custom_onnx – это баланс: отличное качество, нулевая или почти нулевая коллизия, плюс скорость у ONNX-модели почти как у простых алгоритмов.

Если составить график, то можно увидеть, что базовая реализация object hashcode показывает стабильность на всех тестах. На рисунке 17 представлен график времени выполнения.

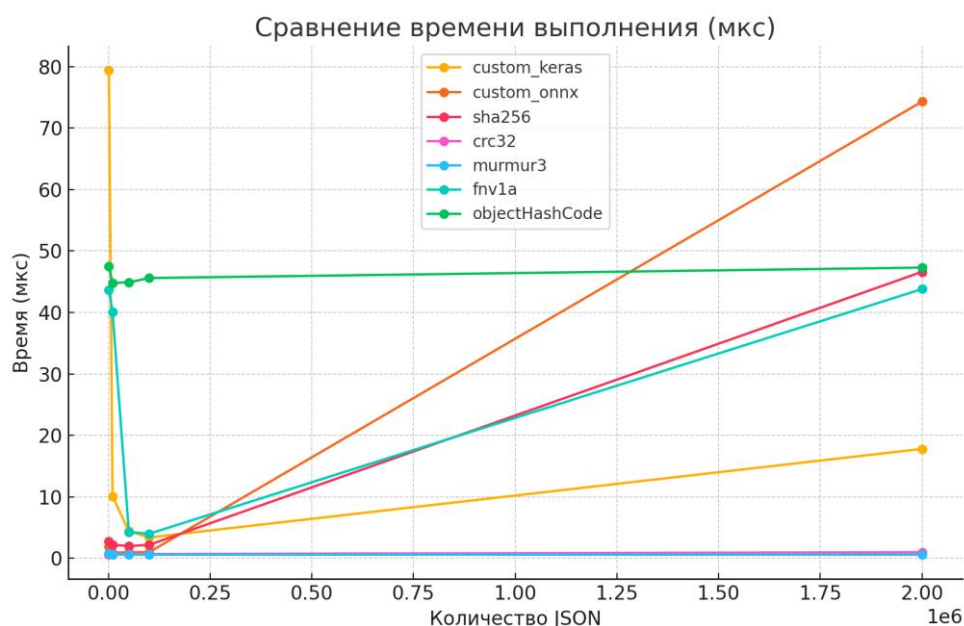


Рисунок 17 - Время выполнения функций

На рисунке 18 приведен график коллизий функций.

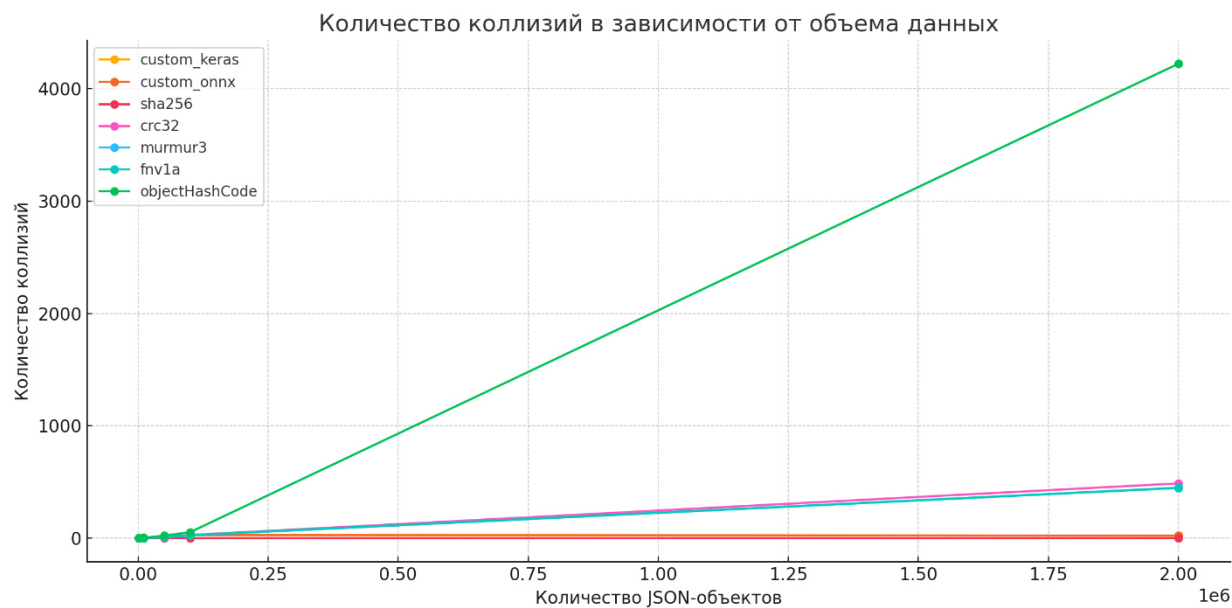


Рисунок 18 – График коллизий функций

Из графика можно увидеть, что при увеличении данных базовый алгоритм языка Java выдает все больше коллизий, самый лучший результат у SHA256, но разработанный ИИ выдает вполне подходящий результат.

ЗАКЛЮЧЕНИЕ

В процессе выполнения выпускной квалификационной работы была спроектирована и реализована интеллектуальная система для автоматизированного выбора оптимального алгоритма хеширования JSON-объектов, тем самым была решена задача оптимизации хеширования в языке Java и не только. Работа включала как исследовательскую, так и практическую составляющие и охватывала весь цикл разработки программного обеспечения – от анализа предметной области до тестирования готовой системы.

Были последовательно решены следующие задачи:

- проведён обзор криптографических и некриптографических хеш-функций, оценены их свойства, производительность и применимость к JSON-данным,
- выбраны и обоснованы инструменты реализации, включая Java и Spring Boot для серверной части, Python с библиотеками машинного обучения, инструменты для хранения данных,
- разработана архитектура приложения, реализовано взаимодействие компонентов по REST API, интегрирован модуль искусственного интеллекта с использованием ONNX-моделей,
- обучены две нейросетевые модели, такие как классификатор для выбора хеш-функции и генератор собственного хеша на основе структуры JSON,
- выполнено тестирование системы на различных объёмах данных (от 1 000 до 2 000 000 JSON-объектов), по результатам которого показано преимущество разработанной нейросетевой модели в показателях уникальности и скорости генерации хешей по сравнению со стандартными алгоритмами,
- создан удобный визуальный интерфейс для демонстрации работы сервиса.

Результаты тестирования подтвердили, что предложенный подход обеспечивает высокое качество хеширования JSON-объектов, демонстрируя нулевой или минимальный уровень коллизий даже на больших объёмах данных. Использование ONNX-моделей позволило достичь высокой производительности при инференсе, сохранив переносимость между Python и Java.

Разработанная система может использоваться как в составе более крупных платформ, работающих с JSON-данными (например, для валидации, контроля целостности или дедупликации), так и как самостоятельный сервис. В будущем возможна доработка и расширение функционала, включая поддержку авторизации, хранение истории запросов и реализацию более сложных моделей генерации хешей.

Таким образом, цели выпускной квалификационной работы достигнуты, система полностью реализована и продемонстрировала свою практическую состоятельность.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Кнутис Д. Искусство программирования. Т. 3: Сортировка и поиск. – М.: Вильямс, 2021. – 912 с.
2. Блох Дж. Эффективное программирование: для Java-разработчиков. – СПб.: Питер, 2020. – 384 с.
3. Чернавский С. В. Современные подходы к построению совершенных хеш-функций // Вестник компьютерных и информационных технологий. – 2019. – № 6. – С. 25–31.
4. Васильев С. И. Теория алгоритмов и хеширования: учеб. пособие. – М.: БИНОМ. Лаборатория знаний, 2020. – 256 с.
5. Menezes A., van Oorschot P., Vanstone S. Handbook of Applied Cryptography. – CRC Press, 1996. – 816 p.
6. ГОСТ Р 34.11–2012. Информационная технология. Криптографическая защита информации. Функция хэширования. – Введ. 01.01.2013.
7. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. FIPS PUB 202. – NIST, 2015. – URL: <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf> (дата обращения: 04.02.2025).
8. Teschner A., Dietrich S. Non-Cryptographic Hash Functions: A Survey and Classification // Journal of Computer Science and Technology. – 2021. – Vol. 36(5). – P. 973–990.
9. Бобров А. С. Эффективность хеширования в структурах данных // Науч.-техн. вестник информационных технологий. – 2020. – № 2. – С. 15–22.
10. Fletcher J. An Arithmetic Checksum for Serial Transmissions // IEEE Transactions on Communications. – 1982. – Vol. 30(1). – P. 247–252.
11. Bloch J. Effective Java. 3rd ed. – Addison-Wesley, 2018. – 416 p.
12. Fowler G., Noll L.C., Vo K. The FNV Non-Cryptographic Hash Algorithm. – URL: <https://tools.ietf.org/html/draft-eastlake-fnv-03> (дата обращения: 14.02.2025).

13. Jenkins B. Survey of Hash Functions for Hash Table Lookup. – URL: <http://www.burtleburtle.net/bob/hash/doobs.html> (дата обращения: 04.03.2025).
14. MurmurHash3 обладает отличными свойствами случайности и очень приличной скоростью (реализация в Unity) // Unity Blog. – URL: <https://unity.com/ru/blog/murmurhash3-performance> (дата обращения: 05.03.2025).
15. Пранкевичюс А. Hash Functions all the way down // блог Aras. – URL: <https://aras-p.info/blog/2016/08/02/Hash-Functions-all-the-way-down> (дата обращения: 08.03.2025).
16. Андреева Е. Как выбрать технологический стек при разработке цифрового сервиса // Uplab. – URL: <https://www.uplab.ru/blog/tehnologicheskij-stek/> (дата обращения: 15.03.2025).
17. Хеклер М. Spring Boot по-быстрому. Создаём облачные приложения на Java и Kotlin. – СПб.: Питер, 2025. – 352 с.
18. Урванов Ф. Spring и Spring Boot: разработка облачных приложений на Java. – М., 2024.
19. Гринберг М. Разработка веб-приложений с использованием Flask на языке Python. – М.: ДМК Пресс, 2016. – 274 с.
20. Добро пожаловать во Flask: документация Flask (рус. перев.) – URL: <https://flask-russian-docs.readthedocs.io/ru/0.10.1/> (дата обращения: 28.03.2025).
21. Spring Boot по-быстрому // Изд-во «Питер». – URL: <https://www.piter.com/collection/java/product/spring-boot-po-bystromu> (дата обращения: 01.04.2025).
22. Гриффитс Д., Гриффитс Д. React. Сборник рецептов = React Cookbook. – СПб.: БХВ-Петербург, 2023. – 528 с. – ISBN 978-5-9775-6839-5.
23. React: что это, его преимущества и особенности // OrbitSoft. – URL: <https://orbitsoft.com/ru/blog/react> (дата обращения: 06.04.2025).
24. React — это JavaScript библиотека для создания динамических пользовательских интерфейсов // RSHB Digital. – URL:

<https://rshbdigital.ru/ekspertiza-i-tehnologii/frontend/react> (дата обращения: 09.04.2025).

25. Хеклер М. Spring Boot по-быстрому. – СПб.: Питер, 2025. – Гл. 10: Современные фронтенд-решения.

26. React испортил веб-разработку // Хабр. – URL: <https://habr.com/ru/companies/vdsina/articles/564788> (дата обращения: 14.04.2025).

27. PostgreSQL: работа с JSON и JSONB // Highload.today. – URL: <https://highload.today/postgresql-json-i-jsonb/> (дата обращения: 28.04.2025).

28. PostgreSQL: соответствие ACID и транзакционная модель // PostgreSQL Manual. – URL: <https://postgrespro.ru/docs/postgresql/current/tutorial-transactions> (дата обращения: 01.05.2025).

29. Кацуба А. С. PostgreSQL для профессионалов: расширенные функции SQL, хранимые процедуры и аналитика. – М.: БХВ-Петербург, 2023. – 320 с.

30. Docker и PostgreSQL: как быстро развернуть СУБД с помощью контейнера // Хабр. – URL: <https://habr.com/ru/articles/537468/> (дата обращения: 03.05.2025).

31. IntelliJ IDEA: обзор и настройка популярной среды разработки // GitVerse.ru. – URL: <https://gitverse.ru/blog/articles/development/560-intellij-idea-chto-eto-za-sreda-razrabotki> (дата обращения: 07.05.2025).

32. Что такое Visual Studio Code и зачем он нужен // Workspace.ru. – URL: <https://workspace.ru/tools/environment/visual-studio-code> (дата обращения: 09.05.2025).

33. Хакимов А. Б. Контейнеризация программных продуктов с использованием Docker // Программная инженерия: теория и практика. – М.: Информатика, 2022. – С. 212–225.

34. Цветков И. В. Docker и микросервисы: практика построения распределённых приложений. – СПб.: Питер, 2023. – 288 с.

ПРИЛОЖЕНИЕ А

(обязательное)

Оптимизация алгоритмов для вычисления хеш-кода

в языке java

Листинг серверной части приложения

ПРИЛОЖЕНИЕ А.1

Файл «index.html»

```
<!DOCTYPE html>
<html lang="ru">
<head>
  <meta charset="UTF-8">
  <title>Оптимизация алгоритмов хеширования и сравнительный
анализ</title>
  <meta name="viewport" content="width=device-width, initial-
scale=1">
  <style>
    body {
      font-family: 'Segoe UI', Arial, sans-serif;
      background: linear-gradient(135deg, #f5f7fa 0%,
#c3cfe2 100%);
      margin: 0;
      padding: 0;
      min-height: 100vh;
    }
    .container {
      max-width: 560px;
      margin: 40px auto;
      background: #fff;
      border-radius: 24px;
      box-shadow: 0 8px 40px rgba(80,100,120,.10);
      padding: 2.7em 2em 2em 2em;
    }
    h2 {
      color: #275bae;
      text-align: center;
      letter-spacing: 1px;
      margin-bottom: 1.7em;
    }
    .section-title {
      color: #3564b5;
      font-size: 1.18em;
      margin-bottom: 0.6em;
      margin-top: 2em;
    }
    label {
      font-weight: 600;
      display: block;
      margin-bottom: 0.5em;
      color: #4f5e72;
    }
    textarea {
      width: 100%;
      height: 120px;
      border-radius: 10px;
      border: 1.3px solid #c7d0db;
      padding: 1em;
```

```

        font-size: 1.05em;
        margin-bottom: 1.2em;
        background: #f4f7fb;
        resize: vertical;
        box-sizing: border-box;
        transition: border .3s;
    }
    textarea:focus {
        border: 1.5px solid #88a6ec;
        outline: none;
    }
    button {
        display: block;
        width: 100%;
        padding: 1em;
        background: linear-gradient(90deg, #4364f7,
#6fb1fc);
        color: #fff;
        font-size: 1.10em;
        font-weight: bold;
        border: none;
        border-radius: 10px;
        cursor: pointer;
        margin-top: 1em;
        box-shadow: 0 4px 12px rgba(79,94,114,0.09);
        transition: background .22s, box-shadow .22s;
    }
    button:hover {
        background: linear-gradient(90deg, #2d4373,
#4e9efc);
        box-shadow: 0 8px 32px rgba(79,94,114,0.15);
    }
    .result {
        margin-top: 2em;
        background: #f4f7fb;
        border-radius: 12px;
        padding: 1.1em 1em 1.1em 1em;
        box-shadow: 0 2px 12px #e6eaf5;
        font-size: 1.07em;
        color: #374151;
        word-break: break-all;
        animation: fadein .8s;
    }
    .result span {
        font-weight: bold;
        color: #3564b5;
        margin-right: 7px;
    }
    .metrics-card {
        background: #f7fafc;
        border-radius: 12px;
        box-shadow: 0 1px 6px #e3eaf6;
        padding: 1.1em 1em 1.2em 1em;

```

```

        margin-top: 2em;
    }
    .metrics-table {
        width: 100%;
        border-spacing: 0;
        margin-top: 1.1em;
    }
    .metrics-table th, .metrics-table td {
        padding: 8px 7px;
        text-align: center;
        border-bottom: 1px solid #e3eaf6;
    }
    .metrics-table th {
        background: #e8f0fb;
        font-weight: 600;
        color: #254d7d;
    }
    .metrics-table tr:last-child td { border-bottom: none; }
    .metrics-table tr:hover { background: #f0f6ff; }
    .loader {
        color: #3564b5;
        font-size: 1.04em;
        margin: 1.2em 0;
        text-align: center;
        letter-spacing: 1px;
    }
    .inline-input {
        display: inline-block;
        width: 7em;
        padding: 0.4em;
        font-size: 1em;
        border-radius: 6px;
        border: 1px solid #b6bed1;
        margin: 0 0.6em 0 0.5em;
        text-align: right;
    }
    @keyframes fadein {
        0% { opacity: 0; transform: translateY(14px); }
        100% { opacity: 1; transform: translateY(0); }
    }
    @media (max-width: 650px) {
        .container { max-width: 98vw; padding: 1.2em; }
        textarea { font-size: 1em; }
    }
</style>
</head>
<body>
<div class="container">
    <h2>Оптимизация алгоритмов хеширования и сравнительный
    анализ</h2>
    <div>
        <div class="section-title">1. Рекомендация и генерация
        хеша по вашему JSON</div>

```



```

        <label for="jsonInput">Вставьте JSON:</label>
        <textarea id="jsonInput">{ "name": "Test", "value": 123
    }</textarea>
        <button onclick="sendRequest()">Получить
рекомендации</button>
        <div class="result" id="result"
style="display:none"></div>
    </div>
    <div class="metrics-card">
        <div class="section-title">2. Сравнительный бенчмарк
алгоритмов</div>
        <label>
            Сколько JSON сгенерировать для теста:
            <input type="number" class="inline-input"
id="sampleCount" min="100" max="100000" value="10000">
        </label>
        <button style="margin-top:0.5em;"
onclick="loadMetrics()">Запустить бенчмарк</button>
        <div id="metrics" class="loader">—</div>
    </div>
</div>
<script>
    function sendRequest() {
        const input =
document.getElementById('jsonInput').value;
        document.getElementById('result').style.display =
'none';
        let parsed;
        try {
            parsed = JSON.parse(input);
        } catch (e) {
            document.getElementById('result').style.display =
'block';
            document.getElementById('result').innerHTML =
                "<span style='color: #e54b4b;'>Ошибка!</span>
Некорректный JSON.";
            return;
        }
        fetch('/api/hash/recommend', {
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
            body: JSON.stringify({ json: JSON.stringify(parsed)
        })
    })
        .then(res => res.json())
        .then(data => {
            document.getElementById('result').style.display
= 'block';
            document.getElementById('result').innerHTML =
                `<span>Рекомендованный алгоритм:</span>
${data.recommendedAlgorithm ?? '-'}<br>
                <span>Custom Hash:</span> ${data.customHash
?? '-'}<br>

```

```

        <span>Комментарий:</span> ${data.comment ??
    '-'} `;
    })
    .catch(err => {
        document.getElementById('result').style.display
= 'block';
        document.getElementById('result').innerHTML =
            "<span style='color:
#e54b4b;'>Ошибка!</span> Не удалось обработать запрос.";
    });
}

function loadMetrics() {
    const n = document.getElementById('sampleCount').value;
    document.getElementById('metrics').innerHTML = "<div
class='loader'>Загрузка...</div>";
    fetch('/ai-metrics/benchmark/' + n)
        .then(res => res.json())
        .then(data => {
            const results = data.results;
            if (!results) {
                document.getElementById('metrics').innerHTML
= "<div class='loader'>Данные отсутствуют</div>";
                return;
            }
            let html = `<table class="metrics-table">
                <tr>
                    <th>Алгоритм</th><th>Уникальных</th><th>Коллизий</th><th>Время
                    (мкс)</th>
                </tr>`;
            for (const key in results) {
                const r = results[key];
                html += `<tr>
                    <td>${key}</td>
                    <td>${r.unique}</td>
                    <td>${r.collisions}</td>
                    <td>${r.mean_time.toFixed(1)}</td>
                </tr>`;
            }
            html += "</table>";
            document.getElementById('metrics').innerHTML =
html;
        });
}
</script>
</body>
</html>

```

ПРИЛОЖЕНИЕ А.2

Файл «AIMetricsController.java»

```
@RestController
@RequestMapping("/ai-metrics")
public class AIMetricsController {

    private final AIMetricsService aiMetricsService;

    public AIMetricsController(AIMetricsService
aiMetricsService) {
        this.aiMetricsService = aiMetricsService;
    }

    @GetMapping("/classifier")
    public JsonNode getClassifierMetrics() throws Exception {
        return aiMetricsService.getClassifierMetrics();
    }

    @GetMapping("/custom-hash")
    public JsonNode getCustomHashMetrics() throws Exception {
        return aiMetricsService.getCustomHashMetrics();
    }

    @GetMapping("/benchmark/{nSamples}")
    public BenchmarkResponse getCustomBenchmark(@PathVariable
int nSamples) {
        return aiMetricsService.getMetrics(nSamples);
    }
}
```

ПРИЛОЖЕНИЕ А.3

Файл «AIMetricsService.java»

```
@Service
public class AIMetricsService {

    private static final String BASE_URL =
"http://localhost:8000";
    private final ObjectMapper objectMapper = new
ObjectMapper();
    private final HttpClient client =
HttpClient.newHttpClient();
    private final RestTemplate restTemplate = new
RestTemplate();

    public JsonNode getClassifierMetrics() throws Exception {
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(BASE_URL +
"/metrics/classifier"))
            .GET()
            .build();
        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        return objectMapper.readTree(response.body());
    }

    public JsonNode getCustomHashMetrics() throws Exception {
        HttpRequest request = HttpRequest.newBuilder()
            .uri(URI.create(BASE_URL +
"/metrics/custom_hash"))
            .GET()
            .build();
        HttpResponse<String> response = client.send(request,
HttpResponse.BodyHandlers.ofString());
        return objectMapper.readTree(response.body());
    }

    public BenchmarkResponse getMetrics(int nSamples) {
        String url = "http://localhost:8000/benchmark";
        Map<String, Integer> body = new HashMap<>();
        body.put("n_samples", nSamples);

        HttpEntity<Map<String, Integer>> entity = new
HttpEntity<>(body);
        ResponseEntity<BenchmarkResponse> response =
restTemplate
            .postForEntity(url, entity,
BenchmarkResponse.class);
        return response.getBody();
    }
}
```


ПРИЛОЖЕНИЕ А.4

Файл «RecommendationService.java»

```
@Service
public class RecommendationService {

    public RecommendationResponse getAiRecommendation(String
json) {
        RestTemplate restTemplate = new RestTemplate();
        String url = "http://localhost:8000/predict";
        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_JSON);

        Map<String, String> req = new HashMap<>();
        req.put("json", json);

        HttpEntity<Map<String, String>> entity = new
HttpEntity<>(req, headers);

        ResponseEntity<RecommendationResponse> response =
            restTemplate.postForEntity(url, entity,
RecommendationResponse.class);

        return response.getBody();
    }
}
```

ПРИЛОЖЕНИЕ Б

(обязательное)

Оптимизация алгоритмов для вычисления хеш-кода

в языке java

Листинг модуля искусственного интеллекта

ПРИЛОЖЕНИЕ Б.1

Файл «Main.py»

```
import json
import random
import string
import time
import numpy as np
import tensorflow as tf
import hashlib
import joblib
import zlib
import mmh3
import onnxruntime as ort

from datetime import datetime, timedelta

from fastapi import FastAPI, Request
from pydantic import BaseModel

app = FastAPI()

clf_model = joblib.load("hash_ai_model.joblib")

# Загрузка custom hash-модели (keras)
custom_model =
tf.keras.models.load_model("custom_hash_model.h5",
compile=False)

# Загрузка custom hash-модели (onnx)
onnx_session = ort.InferenceSession("custom_hash_model.onnx")

# -- Вспомогательные функции (можно вынести в utils) --

def entropy(values):
    if not values:
        return 0
    values = [v for v in values if v]
    if not values:
        return 0
    unique, counts = np.unique(values, return_counts=True)
    probs = counts / counts.sum()
    return float(-np.sum(probs * np.log2(probs)))

def calc_sha256_8(js_str):
    h = hashlib.sha256(js_str.encode('utf-8')).digest()
    return list(h[:8])

def calc_crc32(js_str):
    return zlib.crc32(js_str.encode('utf-8'))
```

```

def calc_ascii_sum(js_str):
    return sum(ord(c) for c in js_str)

def first_n_bytes(js_str, n=8):
    b = js_str.encode('utf-8')
    return list(b[:n]) + [0]*(n - len(b)) if len(b) < n else
list(b[:n])

def get_features_from_json(json_obj):
    json_str = json.dumps(json_obj, ensure_ascii=False)
    size_bytes = len(json_str.encode("utf-8"))
    def count_keys(obj, depth=1, empty=0, nested_dicts=0):
        if isinstance(obj, dict):
            count = len(obj)
            depths = [depth]
            for v in obj.values():
                if isinstance(v, dict):
                    nested_dicts += 1
            c, d, e, n = count_keys(v, depth+1, empty,
nested_dicts)
            count += c
            depths += d
            empty += e
            nested_dicts = n
            return count, depths, empty, nested_dicts
        elif isinstance(obj, list):
            count = 0
            depths = [depth]
            for item in obj:
                c, d, e, n = count_keys(item, depth+1, empty,
nested_dicts)
                count += c
                depths += d
                empty += e
                nested_dicts = n
            if not obj:
                empty += 1
            return count, depths, empty, nested_dicts
        else:
            if obj in [None, "", []]:
                return 0, [depth], 1, nested_dicts
            return 0, [depth], 0, nested_dicts
    def collect_strs(obj):
        strings = []
        if isinstance(obj, dict):
            for v in obj.values():
                strings += collect_strs(v)
        elif isinstance(obj, list):
            for item in obj:
                strings += collect_strs(item)
        elif isinstance(obj, str):
            strings.append(obj)

```



```

        return strings
def collect_arrays(obj):
    arrays = []
    if isinstance(obj, list):
        arrays.append(len(obj))
        for item in obj:
            arrays += collect_arrays(item)
    elif isinstance(obj, dict):
        for v in obj.values():
            arrays += collect_arrays(v)
    return arrays
num_keys, depths, empty_values, nested_dicts =

def hash_vector_to_hex(hash_vec):
    hash_bytes = bytes([int(abs(x)) % 256 for x in hash_vec])
    return hash_bytes.hex()

def fnv1a_32(s):
    h = 0x811c9dc5
    for c in s.encode('utf-8'):
        h ^= c
        h = (h * 0x01000193) & 0xffffffff
    return h

def object_hash_code(s):
    h = 0
    for c in s:
        h = (31 * h + ord(c)) & 0xFFFFFFFF
    return h

def random_string(length=None):
    if length is None:
        length = random.randint(2, 15)
    alphabet = string.ascii_letters + string.digits + " _-"
    return ''.join(random.choices(alphabet, k=length))
# -- ONNX custom hash инференс --
def custom_hash_onnx(features_np):
    input_name = onnx_session.get_inputs()[0].name
    result = onnx_session.run(None, {input_name:
features_np.astype(np.float32)})[0]
    return result

@app.post("/predict")
async def predict(request: Request):
    data = await request.json()
    try:
        json_obj = json.loads(data["json"])
    except Exception:
        return {"error": "Некорректный JSON"}
    features = get_features_from_json(json_obj)
    features_np = np.array(features).reshape(1, -1)
    predicted = clf_model.predict(features_np)[0]
    # >>> Это важно! <<<

```

```

    if isinstance(predicted, (np.generic, np.ndarray, list)):
        try:
            predicted = predicted.item()
        except Exception:
            predicted = str(predicted)
    return {
        "recommendedAlgorithm": predicted,
        "customHash":
hash_vector_to_hex(custom_model.predict(features_np)[0]),
        "comment": "ИИ выбрал лучший алгоритм и сгенерировал
собственный хеш-код"
    }

# -- API для бенчмарка --
@app.post("/benchmark")
async def api_benchmark(request: Request):
    data = await request.json()
    n_samples = data.get("n_samples", 10000)

    jsons = [random_json() for _ in range(n_samples)]
    json_strs = [json.dumps(js, ensure_ascii=False) for js in
jsons]
    features_arr = np.array([get_features_from_json(js) for js
in jsons])
    json_strs = list(set(json_strs))
    n_samples = len(json_strs)

    # Custom hash (Keras)
    t1 = time.perf_counter()
    hash_vecs_keras = custom_model.predict(features_arr,
batch_size=512, verbose=0)
    t2 = time.perf_counter()
    hashes_custom_keras = set(hash_vector_to_hex(vec) for vec in
hash_vecs_keras)
    custom_time_keras = t2 - t1

    # Custom hash (ONNX)
    t1_onnx = time.perf_counter()
    hash_vecs_onnx = custom_hash_onnx(features_arr)
    t2_onnx = time.perf_counter()
    hashes_custom_onnx = set(hash_vector_to_hex(vec) for vec in
hash_vecs_onnx)
    custom_time_onnx = t2_onnx - t1_onnx

    # SHA-256
    t3 = time.perf_counter()
    hashes_sha = set(hashlib.sha256(js.encode('utf-
8')).hexdigest() for js in json_strs)
    t4 = time.perf_counter()
    sha_time = t4 - t3

    # CRC32

```

```

    t5 = time.perf_counter()
    hashes_crc32 = set(zlib.crc32(js.encode('utf-8')) for js in
json_strs)
    t6 = time.perf_counter()
    crc32_time = t6 - t5

    # MurmurHash3
    t7 = time.perf_counter()
    hashes_murmur = set(mmh3.hash(js) for js in json_strs)
    t8 = time.perf_counter()
    murmur_time = t8 - t7

    # FNV-1a
    t9 = time.perf_counter()
    hashes_fnv = set(fnv1a_32(js) for js in json_strs)
    t10 = time.perf_counter()
    fnv_time = t10 - t9

    # Object.hashCode
    t11 = time.perf_counter()
    hashes_obj = set(object_hash_code(js) for js in json_strs)
    t12 = time.perf_counter()
    obj_time = t12 - t11

@app.get("/sample_json")
def sample_json(n: int = 3):
    """Вернуть несколько случайных JSON для проверки
структуры"""
    samples = [random_json() for _ in range(n)]
    return samples

```

ПРИЛОЖЕНИЕ Б.2

Файл «train_custom_hash.py»

```
import numpy as np
import pandas as pd
import tensorflow as tf
import hashlib

df = pd.read_csv("dataset_real.csv")

# Фильтрация: оставляем только те строки, где json_str не пустой
# и это именно строка
df = df[df["json_str"].notnull()]
df = df[df["json_str"].apply(lambda x: isinstance(x, str))]

print(f"Датасет после фильтрации: {len(df)} строк")

X = df.drop(columns=["bestAlgorithm", "json_str"]).values

def str_to_hash_vec(js_str):
    # Если по каким-то причинам всё равно не строка — скипнем
    if not isinstance(js_str, str):
        return np.zeros(16, dtype=np.uint8)
    h = hashlib.sha256(js_str.encode('utf-8')).digest()
    return np.frombuffer(h[:16], dtype=np.uint8)

# Генерируем y только для валидных строк
y = np.stack([str_to_hash_vec(js) for js in df["json_str"]])

print(f"X shape: {X.shape}, y shape: {y.shape}")

model = tf.keras.Sequential([
    tf.keras.layers.Input(shape=(X.shape[1],)),
    tf.keras.layers.BatchNormalization(),
    tf.keras.layers.Dense(256, activation='relu'),
    tf.keras.layers.Dropout(0.3),
    tf.keras.layers.Dense(128, activation='relu'),
    tf.keras.layers.Dense(16, activation='linear')
])

model.compile(optimizer='adam', loss='mae')
model.fit(X, y, epochs=50, batch_size=64, validation_split=0.2)
model.save("custom_hash_model.h5")
print("CustomHash модель сохранена!")
```


ПРИЛОЖЕНИЕ В

(обязательное)

Оптимизация алгоритмов для вычисления хеш-кода

в языке java

Презентация

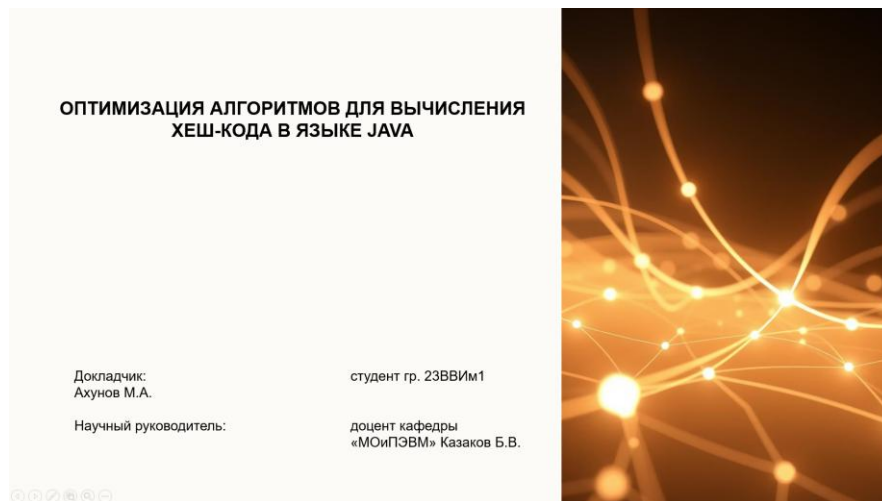


Рисунок В.1 – Фрагмент презентации

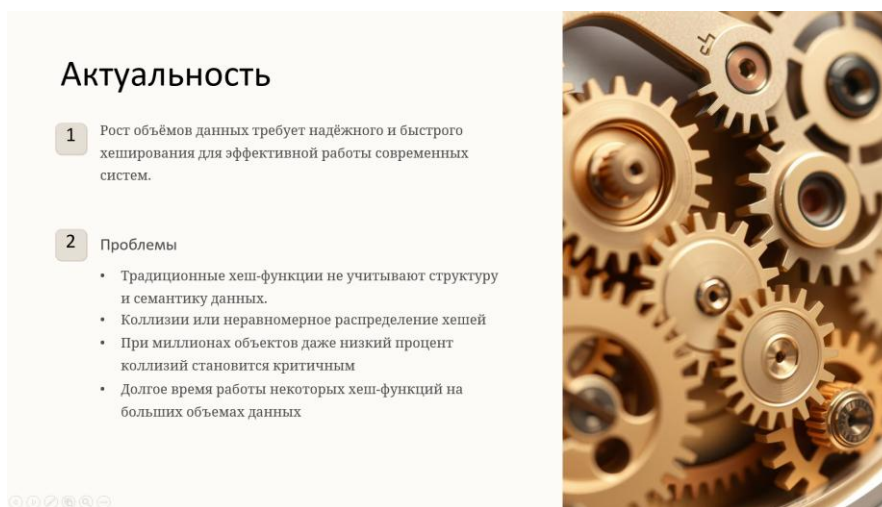


Рисунок В.2 – Фрагмент презентации



Рисунок В.3 – Фрагмент презентации

Сравнение хеш-функций

Алгоритм	Криптостойкость	Равномерность	Вероятность коллизии	Скорость	Сложность реализации	Подходит для JSON
SHA-256	Высокая	Отличная	Незначительная (2^{-128})	Низкая	Высокая	Да (контроль изменений)
ГОСТ Р 34.11-94	Умеренно высокая	Хорошая	Теоретически снижена (2^{-105})	Низкая	Высокая	Да (в гос. стандартах)
FNV-1a	Отсутствует	Средняя	Высокая при больших объёмах	Очень высокая	Очень низкая	Ограничено
MurmurHash3	Отсутствует	Отличная	Низкая для непреднамеренных данных	Очень высокая	Средняя	Да (сравнение)
CRC32	Отсутствует	Хорошая	Низкая случайно, высокая при атаке	Очень высокая	Низкая	Ограничено (целостность)
Object.hashCode	Отсутствует	Псевдослучайная	Зависит от реализации, возможны	Максимальная	Не требуется вручную	Нет (для сравнения содержимого)

Рисунок В.4 – Фрагмент презентации

Выбор ИИ в качестве оптимизации

Java — строго типизированный язык с мощной экосистемой, но встроенные хеш-функции (например, `Object.hashCode()`) зависят от ссылок, а не от содержимого.

Хеши должны быть устойчивыми, уникальными и быстрыми.

Методы машинного обучения позволяют:

- анализировать структуру данных,
 - выбирать подходящий алгоритм,
 - генерировать хеш, адаптированный под входные данные.
- Это делает систему гибкой и более устойчивой к коллизиям.

Рисунок В.5 – Фрагмент презентации

Архитектура Системы

-  Java-сервис
Основа системы, реализован на Spring Boot.
-  Python-модуль ИИ
Включает модели в формате ONNX для производительности.
-  REST API
Обеспечивает взаимодействие между компонентами.
-  Веб-интерфейс
Для удобной работы с JSON-данными.



Рисунок В.6 – Фрагмент презентации

ИИ-Модуль: Сердце Системы

Классификатор	Генератор
Определяет оптимальный хеш-алгоритм для входных данных.	Создает уникальные хеши, учитывая структуру данных.
Обучен на обширном наборе синтетических и реальных JSON.	Минимизирует коллизии для повышения эффективности.
Обе модели постоянно обучаются и улучшаются, адаптируясь к новым типам данных.	



Демонстрация работы программы

1000 JSON

Оптимизация алгоритмов хеширования и сравнительный анализ

1. Рекомендации и генерация ввода по вашему JSON
Введите JSON:

```
{ "name": "test", "value": 123 }
```

[Получить рекомендации](#)

Рекомендуемый алгоритм: Object.hashCode
Custom Hash: `Custom128c0a47678d80d4540338a`
Комментарий: `Object.hashCode` лучше алгоритма в сгенерированном объектном JSON-е

2. Сравнительный бенчмарк алгоритмов
Сколько JSON генерировать для теста:

[Запустить бенчмарк](#)

Алгоритм	Уникальных	Коллизий	Время (мс)
custom_keras	10000	0	10.1
custom_onnx	10000	0	0.9
sha256	10000	0	2.2
crc32	10000	0	0.7
murmur3	10000	0	0.3
fnv1a	10000	0	40.1
objectHashCode	9998	2	44.8

10000 JSON

2. Сравнительный бенчмарк алгоритмов
Сколько JSON генерировать для теста:

[Запустить бенчмарк](#)

Алгоритм	Уникальных	Коллизий	Время (мс)
custom_keras	99974	0	3.4
custom_onnx	99974	0	1.0
sha256	99974	0	2.2
crc32	99973	1	0.7
murmur3	99972	2	0.4
fnv1a	99973	1	40.2
objectHashCode	99948	26	45.6

100000 JSON

2. Сравнительный бенчмарк алгоритмов
Сколько JSON генерировать для теста:

[Запустить бенчмарк](#)

Алгоритм	Уникальных	Коллизий	Время (мс)
custom_keras	99974	0	3.4
custom_onnx	99974	0	1.0
sha256	99974	0	2.2
crc32	99973	1	0.7
murmur3	99972	2	0.4
fnv1a	99973	1	40.2
objectHashCode	99948	26	45.6

Реализованная модель показывает лучшие результаты

105

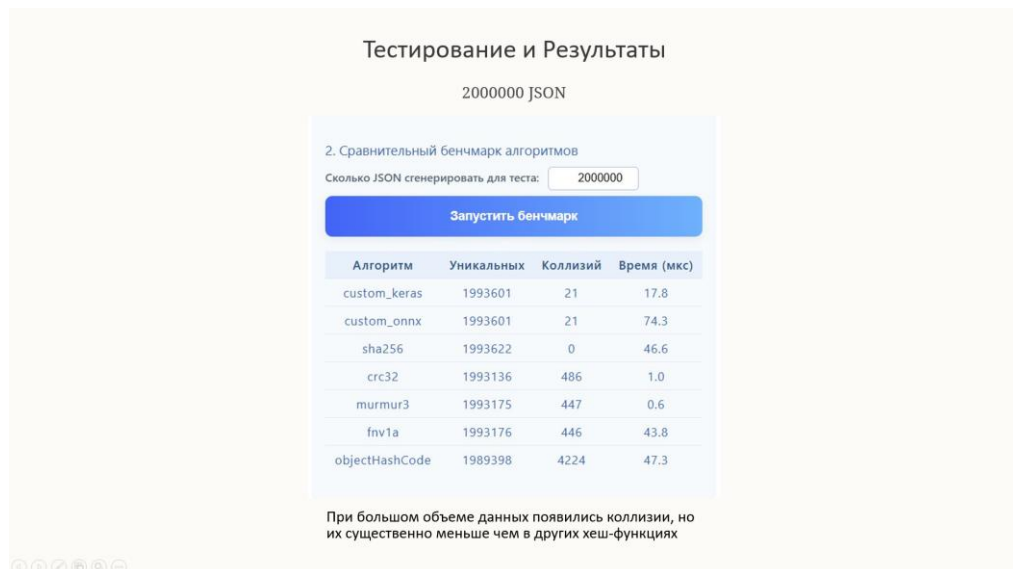


Рисунок В.10 – Фрагмент презентации



Рисунок В.11 – Фрагмент презентации

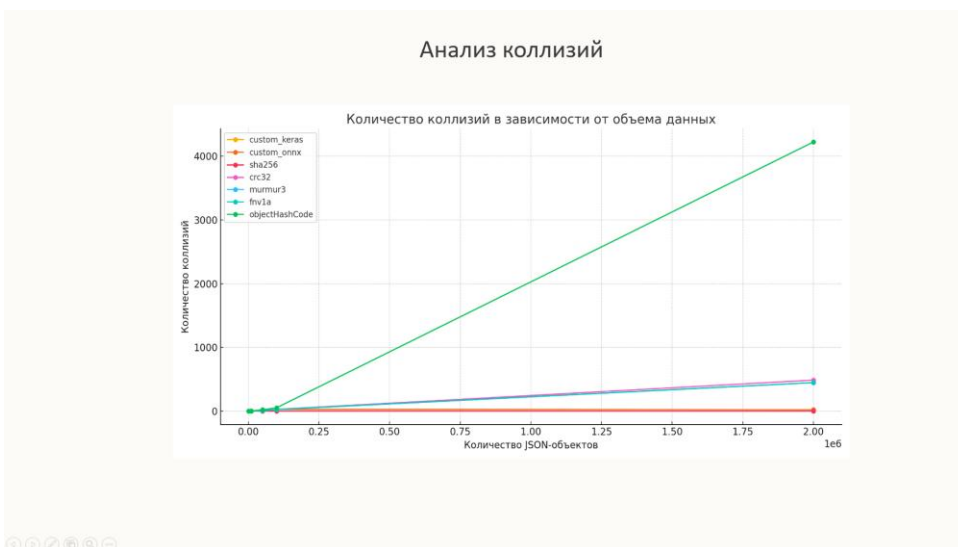


Рисунок В.12 – Фрагмент презентации

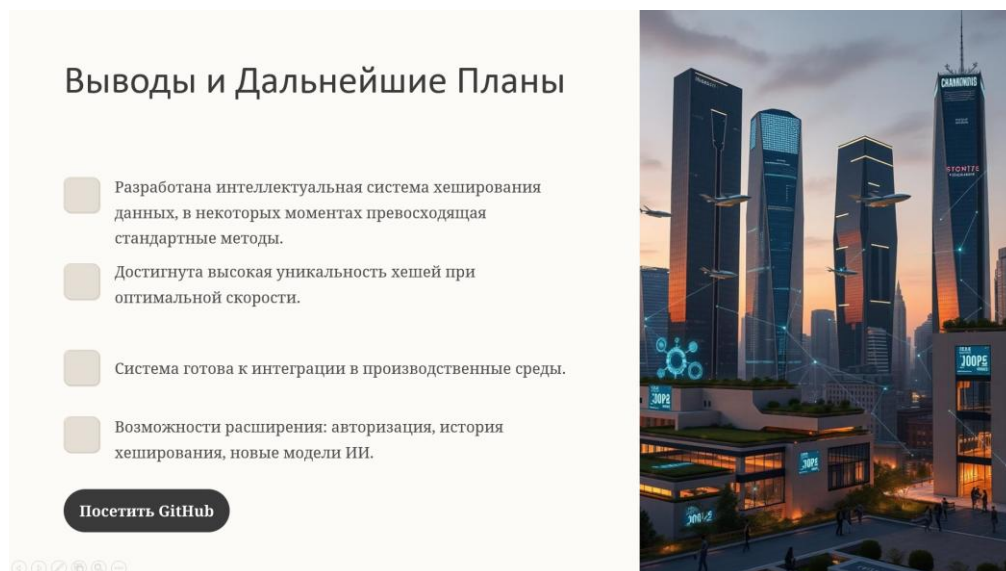


Рисунок В.13 – Фрагмент презентации

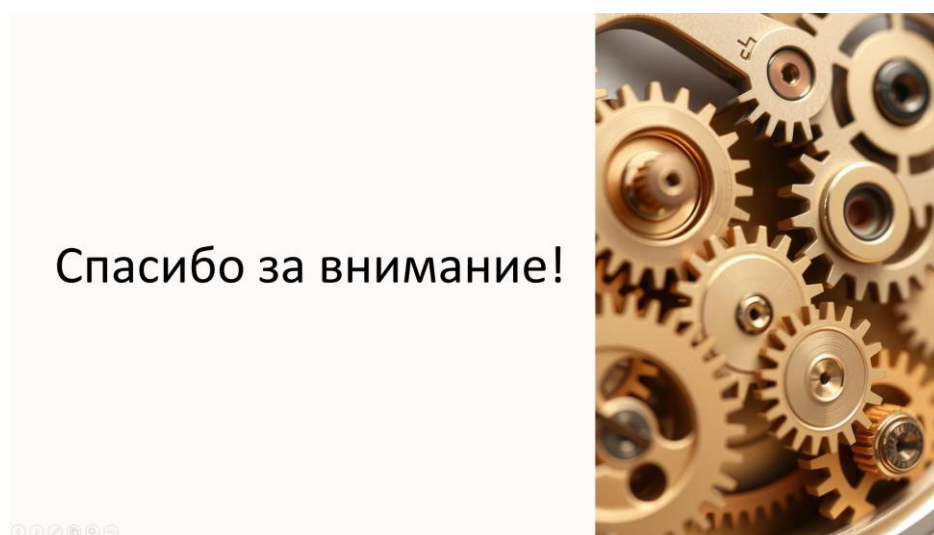


Рисунок В.14 – Фрагмент презентации