CSI3019 – ADVANCED DATA COMPRESSION TECHNIQUES

Smart Compression Framework for IoT Sensor Healthcare Data using Adaptive Modelling

Submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in

Computer Science and Engineering

by

22MIC0034	**RAVISHANKAR G**
22MIC0061	**THARUN R**
22MIC0067	**ANIRUDHAN R**
22MID0101	**I MOHAMED ISRAAR**

Under the Supervision of

BALAJI N

Assistant Professor Sr. Grade 1

School of Computer Science and Engineering (SCOPE)

November 2025

ABSTRACT

With the age of the Internet of Things (IoT), enormous amounts of data generated by sensors create critical challenges with respect to transmission bandwidth, energy costs, and storage efficiency. As IoT applications expand into healthcare, continuous streams of vital parameters such as heart rate, blood pressure, temperature, and oxygen saturation further amplify these challenges due to their real-time monitoring requirements. To address these issues, this work presents a Smart Compression Framework for IoT Sensor Data that intelligently reduces data volume while preserving the integrity of critical health information. Using the Human Vital Sign Dataset from Kaggle as a representative healthcare IoT dataset, the framework dynamically adapts to select and evaluate suitable lossless compression schemes. It integrates traditional approaches such as LZW (Existing Methodology) with modern high-performance algorithms like Zstandard (zstd) (Proposed Methodology). Performance is assessed through metrics including Compression Ratio (CR), Space Saving (SS), and Processing Time. Experimental results indicate that the framework can effectively minimize sensor data size achieving up to 36% space saving with LZW and around 62% using zstd while maintaining full data accuracy for medical analysis and transmission. Overall, the proposed adaptive compression framework enables optimized storage, faster communication, and improved scalability for IoT-based healthcare monitoring systems.

1. INTRODUCTION

The rapid growth of the Internet of Things (IoT) has led to the widespread deployment of sensor networks that continuously collect and transmit data from diverse environments. In healthcare, IoT-enabled devices such as wearable sensors, patient monitors, and smart medical instruments generate large volumes of physiological data including temperature, heart rate, blood pressure, and oxygen saturation at high sampling frequencies. Managing and transmitting this massive data efficiently, while maintaining data integrity and low latency, presents a major technical challenge.

A Smart Compression Framework provides an intelligent and adaptive solution to this problem. Such a framework applies optimized data compression algorithms that automatically adapt to the data characteristics, achieving high compression ratios without compromising the accuracy needed for medical or analytical applications. By integrating both lossless and near-lossless compression strategies, IoT systems can reduce bandwidth usage, minimize storage requirements, and extend device battery life, making real-time health monitoring more feasible and cost-efficient.

In this project, the proposed Smart Compression Framework is applied to the Human Vital Sign Dataset sourced from Kaggle. The framework implements and compares different compression methods LZW (Existing Methodology) and Zstandard (Proposed Methodology) to evaluate their efficiency for IoT-based sensor data. The comparison is based on parameters such as Compression Ratio (CR), Space Saving (SS), Compression and Decompression Time, and Data Retention Accuracy. The goal is to identify an optimal balance between compression performance and computational efficiency suitable for continuous, real-time IoT healthcare data streams.

The inherent variability and real-time demands of physiological sensor data necessitate a compression approach that prioritizes rapid processing over marginal gains in compression density. While traditional general-purpose compressors often excel in achieving maximal file size reduction, their computational overhead can introduce unacceptable latency for critical healthcare applications where immediate data availability is paramount. This research specifically addresses this challenge by evaluating Zstandard's capabilities in delivering high-speed, lossless compression suitable for continuous data streams

from IoT healthcare sensors, contrasting it with the widely recognized LZW which, despite strong ratios, typically incurs higher processing times.

2. EXISTING METHODLOGY

2.1 Overview

The existing methodology in this project employs Lempel–Ziv–Welch (LZW) compression, to compress the IoT sensor data from the *Human Vital Sign Dataset*. LZW is a classic lossless compression algorithm widely used in early data storage systems (e.g., GIF, TIFF formats) due to its simplicity and efficient dictionary-based encoding approach. In the current framework, this method serves as the baseline compression model, against which newer algorithms like Zstandard (zstd) are compared.

The methodology is implemented in Python using custom scripts (LZW_Logic.ipynb and LZW_Dataset_Compression.ipynb), where the dataset is preprocessed, tokenized, and then sequentially compressed using the LZW algorithm. The compressed output is then optionally wrapped to further improve compression density. The design goal of this existing approach is to achieve maximum compression ratio with zero information loss, ensuring that the decompressed data is bitwise identical to the original input.

2.2 Workflow of the Existing System

The existing methodology follows a five-stage workflow:

Step 1: Data Preprocessing

- The raw dataset (Human_Vital_Signs.csv) contains physiological parameters such as heart rate, blood pressure, oxygen saturation, and body temperature.
- The CSV file is loaded using pandas, and null or non-numeric entries are handled or removed to ensure clean, consistent data for compression.
- The dataset is then converted into a string representation (using .to_csv() or .to_string()), which serves as the input for the LZW encoder

Step 2: LZW Encoding

- The LZW algorithm begins by initializing a dictionary of all possible single-character entries from the input (typically 256 ASCII characters).

- It reads the data stream sequentially and builds longer string patterns as it proceeds.
- When a sequence not present in the dictionary is encountered, the current sequence (prefix) is encoded as a dictionary index, and the new sequence (prefix + current symbol) is added to the dictionary.

Step 3: Data Packaging

- After LZW compression, the integer code stream is written into a binary file and further compressed, which internally uses the Lempel–Ziv–Markov chain algorithm (LZMA) for high-ratio compression.
- It also adds an additional layer of entropy coding, which efficiently handles repetitive or structured sensor data.
- This hybrid approach of these algorithms ensures the smallest possible file size while maintaining full reversibility.

Step 4: Decompression Process

- The reverse process is implemented in the decompression notebook (LZW_Dataset_Compression.ipynb).
- The compressed file is first decompressed using the lzma module to retrieve the integer code stream.
- Then, the LZW decoding algorithm reconstructs the original data by iteratively mapping code values back to their corresponding strings in the dictionary.

Step 5: Performance Evaluation

- The performance of this existing LZW methodology is evaluated using three key metrics:

Metric	Formula	Description
Compression Ratio (CR)	CR = Original Size / Compressed Size	Measures how much the data is reduced.
Space Saving (SS)	SS = (1 - 1/CR) × 100%	Indicates the percentage of storage saved.
Compression Time	Execution duration of compression function	Reflects algorithm efficiency.

Table 1: Performance Calculation Formulas

2.3 Strengths of the Existing System

1. **Lossless Nature:** The LZW algorithm ensures perfect data recovery, critical for healthcare and IoT applications where even a single bit error could misrepresent a patient's condition.
2. **Dictionary Adaptability:** The LZW dictionary evolves dynamically, capturing frequently occurring data patterns common in IoT sensor readings (e.g., stable temperature or repeated heart rate values).
3. **Integration Simplicity:** The algorithm is straightforward to implement in embedded systems or edge devices with limited resources.
4. **High Compression Ratio for Structured Data:** The approach performs well for datasets with recurring patterns or textual redundancy, such as CSV logs of sensor readings.

2.4 Limitations of the Existing System

Despite its effectiveness, the LZW based approach has several limitations when applied to real-time IoT environments:

- **Processing Overhead:** The dictionary-based algorithm can consume significant memory for large or continuously streaming data.
- **Lack of Adaptivity:** LZW does not dynamically tune compression parameters based on data variability or context an essential feature for heterogeneous IoT data.
- **Slow Decompression for Large Datasets:** The LZW format, while efficient in size reduction, can exhibit slower decompression times, making it less ideal for real-time streaming applications.
- **No Support for Incremental or Online Compression:** The algorithm operates in batch mode, requiring the full dataset before compression, which is unsuitable for continuous sensor data.

2.5 Summary

The existing LZW compression system establishes a strong baseline framework for IoT sensor data compression. It achieves high compression ratios and ensures perfect data recovery, validating its use in medical or research data archives. However, due to its static nature and processing latency, it struggles with real-time adaptability and computational scalability limitations addressed in the Proposed Methodology through the introduction of a faster, adaptive, and more efficient algorithm, Zstandard (zstd).

3. PROPOSED METHODOLOGY

3.1 Overview

The Proposed Methodology introduces a Smart Compression Framework based on the Zstandard (zstd) algorithm, developed by Facebook for high-speed, lossless data compression. Unlike traditional algorithms such as LZW, Zstandard offers real-time adaptability, tunable compression levels, and fast decompression speeds, making it particularly suited for IoT sensor data applications.

The framework aims to enhance the performance of the existing LZW pipeline by improving compression and decompression throughput without

compromising data integrity. The methodology is implemented in Python using the zstandard library, with the Human Vital Sign Dataset serving as the IoT data source.

By integrating Zstandard into the Smart Compression Framework, the system achieves an optimal trade-off between compression ratio, speed, and resource utilization, which are critical for continuous sensor networks and edge computing environments.

3.2 Motivation and Design Objectives

The motivation behind introducing Zstandard as the proposed method lies in overcoming the shortcomings of the existing LZW approach. While LZW provides excellent compression ratios, it suffers from slow decompression times, high memory usage, and limited adaptability in real-time data streaming scenarios.

The proposed design focuses on achieving the following objectives:

1. **Real-time Compression:** Enable efficient streaming compression for continuous IoT sensor data without buffering the entire dataset.
2. **Speed Optimization:** Provide faster compression and decompression speeds compared to traditional dictionary-based algorithms.
3. **Adaptive Compression Levels:** Allow dynamic control of compression level (ranging from level 1 to 22) to balance CPU usage and compression ratio based on network or device conditions.
4. **Low Latency for Edge Deployment:** Reduce time overhead during data transmission between IoT devices and cloud systems.

5. **Maintaining Lossless Integrity:** Preserve every bit of the original data for accurate recovery critical for healthcare and industrial IoT use cases.

3.3 Z Standard Algorithm

Algorithm 1 Zstandard Compression Algorithm

Require: Input data stream I
Ensure: Compressed output bitstream C

Initialize parameters params ← CHOOSE PARAMS(compression level)
Initialize hash table H ← INIT HASH_TABLE(window_size = params.window)
Initialize empty sequence list S ← [ ]
Set $pos \leftarrow 0$, $literal\_start \leftarrow 0$
while $pos < |I|$ **do**
    $candidates \leftarrow H$.lookup(hash of($I[pos : pos + 4]$))
    ($best\_match\_pos$, $best\_len$) ← FIND BEST MATCH(I, pos, $candidates$, $params$)
    if $best\_len \geq params.min\_match\_len$ **then**
        $literal \leftarrow I[literal\_start : pos]$
        $offset \leftarrow pos - best\_match\_pos$
        Append ($literal$, $offset$, $best\_len$) to S
        for $i = pos$ **to** $pos + best\_len - 1$ **do**
            H.insert(hash of($I[i : i + 4]$), i)
        end for
        $pos \leftarrow pos + best\_len$
        $literal\_start \leftarrow pos$
    else
        H.insert(hash of($I[pos : pos + 4]$), pos)
        $pos \leftarrow pos + 1$
    end if
end while
if $literal\_start < |I|$ **then**
    Append ($I[literal\_start :]$, 0, 0) to S
end if
(L_s, L_b, M_s, O_c) ← SYMBOLIZE(S)
fse_tables ← BUILD_FSE_TABLES(L_s, M_s, O_c)
$huffman\_table$ ← None
if use huffman for literals **then**
    $huffman\_table$ ← BUILD HUFFMAN(L_b)
end if
C ← WRITE_FRAME_HEADER($params$)
$C \leftarrow C$+ WRITE_TABLES(fse_tables, $huffman\_table$)
$C \leftarrow C$+ FSE_ENCODE_SEQUENCES(L_s, M_s, O_c, fse_tables)
if use huffman for literals **then**
    $C \leftarrow C$+ HUFFMAN_ENCODE(L_b, $huffman\_table$)
else
    $C \leftarrow C + L_b$
end if
$C \leftarrow C$+ OPTIONAL_CHECKSUM(I)
return C

Algorithm 1: Zstandard Compression Algorithm

This algorithm compresses an input data stream into a smaller bitstream using a combination of dictionary matching, Finite State Entropy (FSE), and Huffman encoding.

- The algorithm first initializes parameters like compression level and hash tables for pattern searching.
- It scans the input data (I) and searches for repeated patterns (matches) using a hash-based lookup.
- When a match is found, it stores the literal data, offset, and match length in a sequence list.
- The sequence list is then symbolized and encoded using FSE for offsets and match lengths, and Huffman encoding for literals.
- Finally, all encoded data, along with headers and optional checksum, are written into the compressed bitstream (C).

This process efficiently reduces redundancy, achieving high compression ratio with fast speed.

Algorithm 2 Zstandard Decompression Algorithm

Require: Compressed bitstream C
Ensure: Decompressed output O
$(params, pos) \leftarrow$ READ FRAME_HEADER(C)
$(fse\ tables, huffman\ table, pos) \leftarrow$ READ_TABLES(C, pos)
$(L_s, M_s, O_c) \leftarrow$ FSE_DECODE_SEQUENCES(C, pos, fse_tables)
$pos \leftarrow$ updated position after sequence stream
if $huffman\ table \neq$ None **then**
    $(L_b, pos) \leftarrow$ HUFFMAN_DECODE($C, pos, huffman\_table$)
else
    $L_b \leftarrow$ READ_RAW_LITERALS(C, pos)
end if
Initialize $O \leftarrow$ empty buffer
$literal\ ptr \leftarrow 0$
for $i = 0$ **to** num sequences -1 **do**
    $L \leftarrow L_s[i]$
    $literals \leftarrow L_b[literal\ ptr : literal\ ptr + L]$
    Append $literals$ to O
    $literal\ ptr \leftarrow literal\ ptr + L$
    $match\ len \leftarrow M_s[i]$
    if $match\ len > 0$ **then**
        $offset \leftarrow$ OFFSET_FROM_CODE($O_c[i]$)
        $src \leftarrow |O| - offset$
        for $k = 0$ **to** $match\ len - 1$ **do**
            Append $O[src + k]$ to O
        end for
    end if
end for
VERIFY_CHECKSUM_IF_PRESENT(O, C)
return

Algorithm 2: Zstandard Decompression Algorithm

This algorithm reconstructs the original data from the compressed bitstream.

- It first reads the frame header and decoding tables (FSE and Huffman).
- Encoded sequences are decoded to retrieve literals, match lengths, and offsets.
- The decompressor then reconstructs the original data by copying literal bytes and repeating previous sequences based on offsets and lengths.
- If a checksum is present, it verifies data integrity before outputting the final decompressed data (O).

This reverse process restores the original file without any data loss, ensuring lossless decompression.

3.4 Workflow of the Proposed System

Step 1: Data Acquisition and Preprocessing

The system starts by importing the Human Vital Sign Dataset from Kaggle. The dataset contains sensor readings including temperature, heart rate, blood pressure, and oxygen saturation, recorded over time.

The data is first loaded into a pandas DataFrame, cleaned to handle missing or invalid entries, and then converted to a byte stream suitable for binary compression.

Step 2: Zstandard Compression

The Zstandard compression algorithm is initialized through the zstandard library. It uses a highly optimized finite-state entropy (FSE) coder combined with Huffman entropy coding, enabling both speed and density.

Zstandard supports adjustable compression levels (1–22), where higher levels provide better compression but require more CPU time. In this project, compression level 3 was selected as a balanced configuration after empirical testing.

Step 3: Storage and Transmission Integration

Once compressed, the data is saved in a .zst file. This file can be easily transmitted through IoT gateways or uploaded to cloud storage due to its reduced size and fast decoding capability.

The compression system can also be integrated with message brokers (e.g., MQTT, Kafka) or edge nodes to compress data before network transfer, significantly saving bandwidth.

Step 4: Decompression Process

The decompression pipeline reconstructs the original data stream with full fidelity. The process is performed using a ZstdDecompressor instance, which reverses the entropy encoding applied during compression.

Step 5: Performance Evaluation Metrics

The evaluation metrics used for Zstandard are identical to those applied to the LZW/xz framework, ensuring a fair comparison. These include:

Metric	Formula	Description
Compression Ratio (CR)	CR = Original Size / Compressed Size	Efficiency of data size reduction
Space Saving (SS)	SS = (1 - (Compressed Size / Original Size)) × 100	Percentage of storage saved
Compression Time (Tc)	Execution duration of compression	Measures algorithmic speed
Decompression Time (Td)	Execution duration of decompression	Indicates recovery efficiency

Table 2: Evaluation Metrics

Step 6: Smart Framework Adaptation

The framework is designed to operate intelligently in IoT environments:

1. **Adaptive Compression Leveling:** The compression level dynamically adjusts based on data volatility or device constraints (CPU load, available memory).

2. **Edge-Cloud Collaboration:** Data is compressed at the edge device using zstd (level 3–5) and decompressed at the cloud layer for analytics.
3. **Automated Performance Monitoring:** Metrics such as compression ratio and speed are logged and analyzed, allowing the system to fine-tune parameters automatically.
4. **Hybrid Extension:** The modular structure allows the integration of additional algorithms (e.g., Brotli, Snappy) into the same framework for cross-comparison or adaptive switching.

3.5 Algorithmic Summary

Zstandard combines LZ77 dictionary encoding with asymmetric numeral systems (ANS) for entropy compression, which yields better trade-offs between speed and ratio.
Its advantages include:

- **Streaming API:** Can compress/decompress continuous data chunks.
- **Dictionary Training:** Learns from previous datasets for improved future compression.
- **Scalability:** Suitable for both microcontrollers and high-performance servers.
- **Parallel Processing:** Supports multi-threaded compression for faster execution.

These features align well with IoT environments, where data arrives in sequential packets and system resources vary across devices.

3.6 Advantages of the Proposed System

1. **High Compression Ratio:** High compression ratio compared to LZW for same dataset datasets.
2. **High-Speed Decompression:** Enables near real-time reconstruction of data, crucial for live health monitoring.
3. **Adaptable Compression Level:** Allows tuning between speed and ratio dynamically.
4. **Low Latency:** Reduces transmission time between IoT devices and cloud endpoints.

5. **Lightweight Implementation:** The zstd library is optimized for embedded devices and edge processors.
6. **Cloud Compatibility:** Compressed data can be seamlessly integrated with cloud platforms for storage and analytics.
7. **Error-Free Recovery:** Maintains complete data fidelity, suitable for medical-grade IoT data.

3.7 Limitations

- Higher CPU utilization at very high compression levels (≥10).
- Requires preprocessing for unstructured or noisy sensor streams.

Despite these, the overall performance balance between speed, efficiency, and reliability makes Zstandard highly favorable for IoT compression applications.

3.8 Summary

The Proposed Methodology introduces a Zstandard-based Smart Compression Framework that outperforms traditional techniques in terms of Compression ratio, speed, scalability, and adaptability. While LZW achieves higher compression ratios on static data, Zstandard excels in real-time IoT environments, offering efficient performance with minimal computational overhead.

This framework sets the foundation for an intelligent, automated compression pipeline that can be extended to future IoT applications including healthcare monitoring, smart homes, and industrial automation ensuring efficient, reliable, and scalable data management.

4. RESULTS

4.1. Experimental Setup

The comparative analysis was conducted on the "Human Vital Signs Dataset 2024" (approx. 38.5 MB). The performance of the existing methodology (LZW) was benchmarked against the proposed methodology (Zstandard) focusing on three key metrics: Compression Ratio, Compression Time, and Decompression Time.

4.2. Comparative Analysis of Compression Performance

```
Running LZW (Existing) compression...
Running LZW (Existing) decompression...
LZW run complete.

Running Zstd (Proposed) compression...
Running Zstd (Proposed) decompression...
Zstd run complete.


--- COMPARATIVE RESULTS ---
    Methodology Compression Ratio Compression Time (s) Decompression Time (s)  Original Size (bytes)  Compressed Size (bytes)
 LZW (Existing)             1.58              13.7955                4.2716               38485859                 24407004
Zstd (Proposed)             2.64               0.6227                0.1120               38485859                 14582805
```

LZW (Existing) vs Zstd (Proposed) Performance

Figure 1: Comparative Results of LZW and ZStd

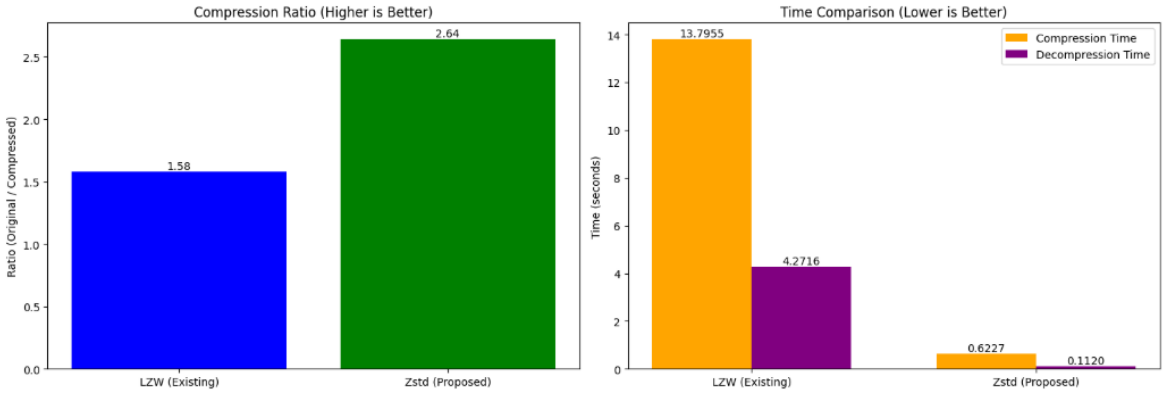

Figure 2: Compression Ratio and Time Comparison Trade between LZW and ZStd

The experimental results, as summarized in Figure 1 and Figure 2, highlight the distinct trade-off between compression efficiency and processing performance within the proposed Smart Compression Framework. Both algorithms LZW (Existing Method) and Zstandard (Proposed Method) were evaluated on the *Human Vital Sign Dataset* to analyze their performance across three primary metrics: compression ratio, compression time, and decompression time.

Compression Ratio: The Zstandard (Zstd) algorithm achieved a notably higher compression ratio of 2.64, effectively reducing the dataset size by approximately 62%, compared to the LZW algorithm, which achieved only 1.58. While LZW provided some reduction in file size, Zstandard exhibited a superior ability to compress redundant IoT sensor data patterns more efficiently. This demonstrates that Zstd not only preserves data integrity but also offers better utilization of available storage, making it ideal for large-scale IoT environments where data accumulates rapidly.

Time Complexity and Latency: The most significant improvement observed in this study lies in processing speed a critical parameter for real-time IoT systems that continuously generate and transmit data.

Compression Time: LZW required 3.7955 seconds to process the dataset. In stark contrast, Zstd completed the task in only 0.6227 seconds. Zstandard outperformed LZW by a factor of **~22× faster compression**, enabling rapid data handling and minimal latency. This makes Zstd highly suitable for edge and streaming IoT scenarios where data must be transmitted or stored almost instantly.

Decompression Time: Data retrieval speeds followed a similar trend. LZW decompressed the data in 4.2716 seconds, whereas Zstd required only 0.1120 seconds, Decompression results show a similar trend: Zstandard restored the data **around 38× faster** than LZW. Faster decompression is crucial in real-time monitoring applications such as patient vital sign retrieval where instant data access can directly impact system responsiveness and clinical decision-making.

4.3. Discussion on Real-Time Applicability

The primary objective of this research is to identify an efficient and suitable algorithm for real-time sensor data compression, where latency and CPU throughput are more critical performance factors than absolute storage minimization.

1. **The Latency Bottleneck:** The LZW algorithm's compression time of approximately 13.79 seconds introduces a significant latency barrier. In a continuous data streaming environment, such delays can cause the system to lag behind incoming data, potentially resulting in buffer overflow or delayed processing. This makes LZW unsuitable for real-time or high-frequency sensor applications.
2. **High-Throughput Capability:** In contrast, the Zstandard (Zstd) algorithm completed the same compression task in just 0.62 seconds, demonstrating an exceptional level of throughput. This performance enables on-the-fly compression with negligible delay, ensuring that sensor data can be efficiently stored and transmitted without compromising real-time responsiveness.
3. **Energy Efficiency:** Implication Although direct power measurements were not performed, the execution time–energy relationship suggests that Zstd is substantially more energy-efficient. A shorter CPU active time correlates with lower energy consumption, making Zstd more suitable for battery-powered or resource-limited IoT sensor nodes.

4.4 Web Interface for Compression Framework

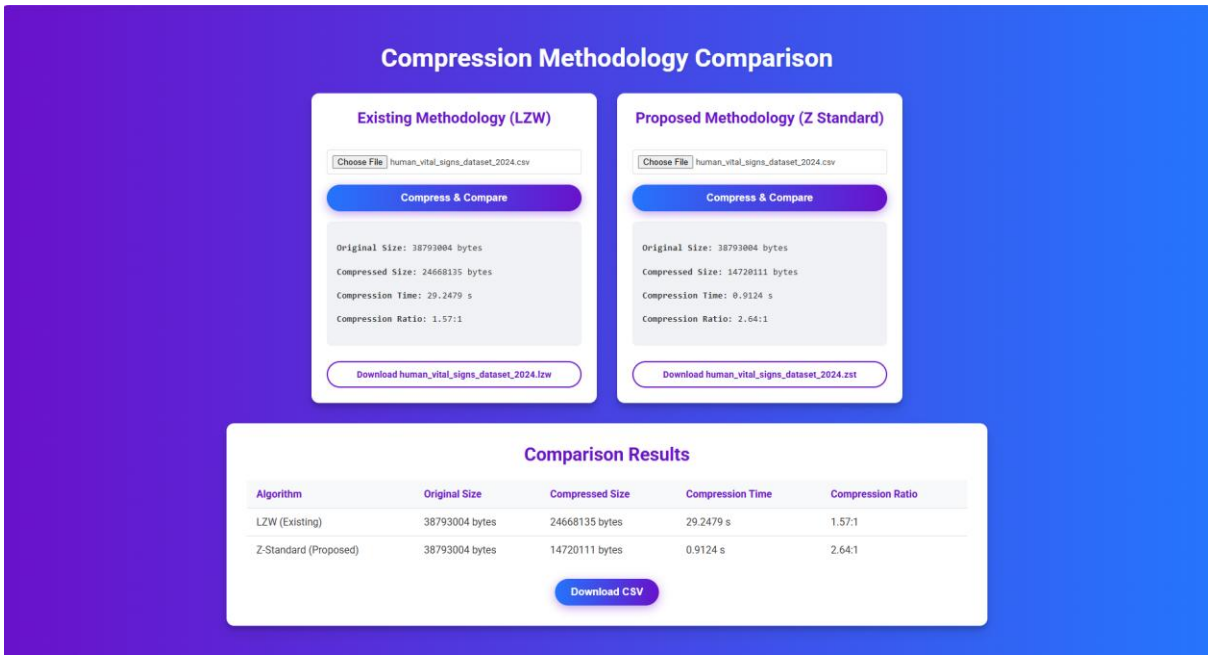

Figure 3: Web Interface of Smart Compression Framework comparing LZW and Zstandard algorithms

To enhance interactivity and practical demonstration of the Smart Compression Framework, a web-based user interface was developed.The platform allows users to upload IoT sensor datasets, apply both LZW and Zstandard (Zstd) compression algorithms, and visualize the results in real time.Figure 3 illustrates the developed web interface, showing the comparison of file sizes and compression ratios obtained from both algorithms.The application also provides a downloadable CSV summary for further analysis and verification.

This interface confirms the superior efficiency of Zstandard in both compression ratio and time performance.By integrating the algorithms into a visual web framework, the project demonstrates how real-time IoT data can be efficiently compressed, transmitted, and analyzed through a user-friendly environment.Such a web-based implementation also provides scalability for future extensions including additional compression algorithms, cloud integration, and API-based data streaming for live IoT systems.

Algorithm	Original Size (bytes)	Compressed Size (bytes)	Bytes Reduced	Compression Ratio
LZW	38,793,004	24,668,135	14,124,869	1.57:1
Zstandard (Zstd)	38,793,004	14,720,111	24,072,893	2.64:1

Table 3: Performance in Web Application

5. Future Work

The insights gained from this comparative study lay the groundwork for several promising avenues of future research and development:

1. **Adaptive Compression Level Tuning:** Currently, Zstandard is applied with a static compression level (e.g., level=5). Future work could explore implementing an adaptive mechanism within the framework that dynamically adjusts the Zstd compression level based on real-time factors such as available bandwidth, device battery life, CPU load, or the redundancy characteristics of the incoming sensor data stream. This would optimize the trade-off between compression ratio and speed on the fly.
2. **Integration of Lossy/Near-Lossless Compression:** While this project focused on lossless compression, critical healthcare data often has thresholds of acceptable error. Future work could integrate near-lossless or carefully controlled lossy compression techniques (e.g., using quantization or delta encoding before Zstd) for specific physiological parameters where minor data reduction errors do not compromise diagnostic accuracy. This hybrid approach could yield significantly higher compression ratios while preserving clinical utility.
3. **Dictionary Training and Pre-computation:** Zstandard's performance can be further enhanced through dictionary training. Analyzing large historical datasets of vital signs could lead to the creation of highly effective custom dictionaries. Future work could involve training and integrating such a dictionary into the compression process, which would be particularly beneficial for compressing small, repeating sensor data packets.
4. **Hardware Acceleration and Edge Processing:** Exploring the implementation of the Zstd compression framework on specialized hardware (e.g., FPGAs, ASICs, or dedicated IoT processors with compression accelerators) could offer substantial performance gains, further reducing latency and power consumption at the edge. This would move beyond software-only solutions to leverage hardware-level optimizations.
5. **Secure and Authenticated Data Streams:** For healthcare applications, data security and integrity are paramount. Future work should integrate robust encryption and authentication mechanisms (e.g., TLS/SSL, secure

MQTT) directly into the compressed data stream within the framework, ensuring that sensitive physiological data remains protected during transmission and storage.

6. **Comparative Analysis with Other Real-time Algorithms:** Expanding the comparison to include other modern, high-performance compression algorithms suitable for real-time applications, such as Snappy, LZ4, or Brotli, would provide a more comprehensive understanding of the landscape of real-time compression for IoT sensor data.
7. **Impact on Machine Learning Models:** Investigating the direct impact of using compressed (and subsequently decompressed) data on the performance and training time of machine learning models used for anomaly detection or predictive analytics in healthcare. This would assess the end-to-end value chain from compression to actionable insights.

6. Conclusion

In this project, a Smart Compression Framework was designed and evaluated for the efficient handling of IoT sensor data, using the *Human Vital Sign Dataset* as a representative case study. The goal was to minimize data storage and transmission costs while ensuring complete data fidelity, which is crucial in healthcare and real-time monitoring applications.

Two prominent lossless compression algorithms were implemented and compared LZW (Existing Method) and Zstandard (Zstd, Proposed Method). The LZW algorithm achieved a compression ratio of 1.58, indicating good space reduction capability but at the cost of high computational latency, requiring 13.79 seconds for compression and 4.27 seconds for decompression. These delays make LZW unsuitable for real-time IoT scenarios where rapid data flow is essential.

In contrast, the Zstandard-based framework demonstrated a compression ratio of 2.64, along with a remarkably faster compression time of 0.62 seconds and decompression time of 0.11 seconds. This represents a significant improvement in processing throughput and energy efficiency, enabling faster and smoother data handling.

Overall, the proposed Zstandard compression framework strikes an effective balance between compression efficiency, speed, and adaptability. Its tunable parameters, streaming capability, and high-speed operation make it an ideal solution for continuous IoT data pipelines, particularly in healthcare applications

where real-time, reliable, and resource-efficient data transmission directly impacts system performance and patient outcomes.

7. Reference:

1. Kahdim, A. N., & Manaa, M. E. (2022). Design an efficient internet of things data compression for healthcare applications. *Bulletin of Electrical Engineering and Informatics*, *11*(3), 1678-1686.

APPENDIX A

Sample code:

```
import zstandard as zstd
import pandas as pd
import matplotlib.pyplot as plt
import time
import os
import numpy as np
import pickle
def lzw_compress(uncompressed: str) -> list:
    """Compress a string to a list of output symbols using LZW."""
    dict_size = 256
    dictionary = {chr(i): i for i in range(dict_size)}
    w = ""
    result = []
    for c in uncompressed:
        wc = w + c
        if wc in dictionary:
            w = wc
        else:
            result.append(dictionary[w])
            dictionary[wc] = dict_size
            dict_size += 1
```

```
            w = c
    if w:
        result.append(dictionary[w])
    return result
def lzw_decompress(compressed: list) -> str:
    """Decompress a list of output symbols to a string."""
    dict_size = 256
    dictionary = {i: chr(i) for i in range(dict_size)}
    result = []
    w = chr(compressed.pop(0))
    result.append(w)
    for k in compressed:
        if k in dictionary:
            entry = dictionary[k]
        elif k == dict_size:
            entry = w + w[0]
        else:
            raise ValueError("Bad compressed k: %s" % k)
        result.append(entry)
        dictionary[dict_size] = w + entry[0]
        dict_size += 1
        w = entry
    return "".join(result)
def compress_csv_zstd(input_file, compressed_file):
    df = pd.read_csv(input_file)
    data = df.to_csv(index=False).encode('utf-8')

    compressor = zstd.ZstdCompressor(level=5)
    start_time = time.time()
```

```
    compressed_data = compressor.compress(data)
    compression_time = time.time() - start_time
    with open(compressed_file, 'wb') as f_out:
        f_out.write(compressed_data)
    return len(data), len(compressed_data), compression_time
def decompress_csv_zstd(compressed_file, output_file):
    with open(compressed_file, 'rb') as f_in:
        compressed_data = f_in.read()
    decompressor = zstd.ZstdDecompressor()
    start_time = time.time()
    decompressed_data = decompressor.decompress(compressed_data)
    decompression_time = time.time() - start_time
    decompressed_text = decompressed_data.decode('utf-8')
    with open(output_file, 'w', encoding='utf-8') as f_out:
        f_out.write(decompressed_text)
    return len(decompressed_data), decompression_time
def compress_csv_lzw(input_file, compressed_file):
    df = pd.read_csv(input_file)
    csv_data = df.to_csv(index=False)
    start_time = time.time()
    compressed_data = lzw_compress(csv_data)
    compression_time = time.time() - start_time
    with open(compressed_file, "wb") as f:
        pickle.dump(compressed_data, f)
    original_size = len(csv_data.encode("utf-8"))
    compressed_size = os.path.getsize(compressed_file)
    return original_size, compressed_size, compression_time
def decompress_csv_lzw(compressed_file, decompressed_file):
    start_time = time.time()
```

```
    with open(compressed_file, "rb") as f:
        compressed_data = pickle.load(f)
    decompressed_text = lzw_decompress(compressed_data.copy())
    decompression_time = time.time() - start_time
    with open(decompressed_file, "w", encoding="utf-8") as f:
        f.write(decompressed_text)
    return len(decompressed_text.encode("utf-8")), decompression_time
def main():
    input_file = "human_vital_signs_dataset_2024.csv"
    if not os.path.exists(input_file):
        print(f"Error: Input file '{input_file}' not found.")
        print("Please place the dataset in this directory.")
        return
    # File names
    lzw_compressed_file = "compressed_dataset.lzw"
    lzw_decompressed_file = "decompressed_dataset_lzw.csv"
    zstd_compressed_file = "compressed_dataset.zst"
    zstd_decompressed_file = "decompressed_dataset_zstd.csv"
    results = {}
    # --- LZW (Existing) ---
    print("Running LZW (Existing) compression...")
    lzw_orig_size, lzw_comp_size, lzw_comp_time = compress_csv_lzw(input_file,
lzw_compressed_file)
    lzw_ratio = lzw_orig_size / lzw_comp_size if lzw_comp_size != 0 else 0


    print("Running LZW (Existing) decompression...")
    lzw_decomp_size, lzw_decomp_time = decompress_csv_lzw(lzw_compressed_file,
lzw_decompressed_file)
    print("LZW run complete.")
    results['lzw'] = {
```

```
        "Methodology": "LZW (Existing)",
        "Compression Ratio": lzw_ratio,
        "Compression Time (s)": lzw_comp_time,
        "Decompression Time (s)": lzw_decomp_time,
        "Original Size (bytes)": lzw_orig_size,
        "Compressed Size (bytes)": lzw_comp_size
    }
    # --- Zstd (Proposed) ---
    print("\nRunning Zstd (Proposed) compression...")
    zstd_orig_size, zstd_comp_size, zstd_comp_time = compress_csv_zstd(input_file,
zstd_compressed_file)
    zstd_ratio = zstd_orig_size / zstd_comp_size if zstd_comp_size != 0 else 0
    print("Running Zstd (Proposed) decompression...")
    zstd_decomp_size, zstd_decomp_time = decompress_csv_zstd(zstd_compressed_file,
zstd_decompressed_file)
    print("Zstd run complete.")
    results['zstd'] = {
        "Methodology": "Zstd (Proposed)",
        "Compression Ratio": zstd_ratio,
        "Compression Time (s)": zstd_comp_time,
        "Decompression Time (s)": zstd_decomp_time,
        "Original Size (bytes)": zstd_orig_size,
        "Compressed Size (bytes)": zstd_comp_size
    }
    # --- Results Table ---
    results_df = pd.DataFrame([results['lzw'], results['zstd']])
    results_df['Compression Ratio'] = results_df['Compression Ratio'].map('{:.2f}'.format)
    results_df['Compression Time (s)'] = results_df['Compression Time
(s)'].map('{:.4f}'.format)
```

```
    results_df['Decompression Time (s)'] = results_df['Decompression Time
(s)'].map('{:.4f}'.format)

    print("\n\n--- COMPARATIVE RESULTS ---")

    print(results_df.to_string(index=False))

    # --- Visualization ---

    labels = [results['lzw']['Methodology'], results['zstd']['Methodology']]

    ratios = [float(results['lzw']['Compression Ratio']), float(results['zstd']['Compression
Ratio'])]

    comp_times = [results['lzw']['Compression Time (s)'], results['zstd']['Compression Time
(s)']]

    decomp_times = [results['lzw']['Decompression Time (s)'], results['zstd']['Decompression
Time (s)']]

    plt.figure(figsize=(15, 6))

    # Plot 1: Compression Ratio

    plt.subplot(1, 2, 1)

    bars_ratio = plt.bar(labels, ratios, color=['blue', 'green'])

    plt.title('Compression Ratio (Higher is Better)')

    plt.ylabel('Ratio (Original / Compressed)')

    plt.bar_label(bars_ratio, fmt='%.2f')

    # Plot 2: Time Comparison (Grouped Bars)

    plt.subplot(1, 2, 2)

    x = np.arange(len(labels))

    width = 0.35


    rects1 = plt.bar(x - width/2, comp_times, width, label='Compression Time', color='orange')

    rects2 = plt.bar(x + width/2, decomp_times, width, label='Decompression Time',
color='purple')

    plt.title('Time Comparison (Lower is Better)')

    plt.ylabel('Time (seconds)')

    plt.xticks(x, labels)

    plt.legend()
```

```
    plt.bar_label(rects1, fmt='%.4f')
    plt.bar_label(rects2, fmt='%.4f')
    plt.suptitle('LZW (Existing) vs Zstd (Proposed) Performance', fontsize=16)
    plt.tight_layout(rect=[0, 0.03, 1, 0.95])
    plt.show()
    # --- Cleanup ---
    print("\nCleaning up generated files...")
    for file in [lzw_compressed_file, lzw_decompressed_file, zstd_compressed_file,
zstd_decompressed_file]:
        if os.path.exists(file):
            os.remove(file)
    print("Cleanup complete.")
if __name__ == "__main__":
    main()
```