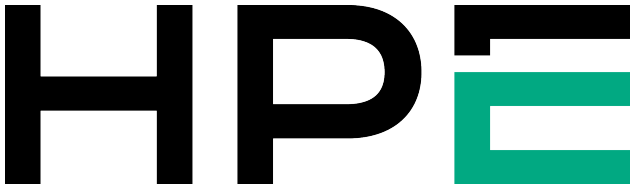
Hewlett Packard Enterprise

CPP 2025



ResilioZ - Backup and Recovery System

Project Report (ID: 62)

Ankur Majumdar

Lalith Dutt Thungala

Mukundh P L

Nykaj A K

Vishnu Narayanan Vinodkumar

HPE Mentor: Sunny Shivam

College Mentor: Dr. Selvi C

Indian Institute of Information Technology, Kottayam

30.06.2025

Contents

Abstract

ResilioZ is a file-level backup and recovery system for Linux, engineered to provide secure, space-efficient, and reliable data protection. The system supports full, incremental, and differential backup strategies through content-defined chunking (FastCDC), deduplication, Zstandard compression, and AES-256 encryption. To ensure data integrity and trustworthiness, each file's content is verified using SHA256 checksums during backup verification and restore operations. The solution is designed to work seamlessly across local filesystems, NFS-mounted directories, and remote SSH/SFTP servers, allowing flexible deployment in diverse environments.

ResilioZ adopts a modular architecture, encompassing components for backup, restore, repository management, and automated scheduling. The Repository Service abstracts storage operations and enforces password-based access control. Complementing this, a centralized repository metadata registry improves performance during access verification. The scheduler operates as a background daemon using cron expressions to automate routine backups.

ResilioZ offers both a Command-Line Interface (CLI) and a Graphical User Interface (GUI) built with Qt, allowing users to interact with the system according to their preference or familiarity. By integrating proven cryptographic, compression, and scheduling libraries, it ensures secure, efficient, and transparent operation. The system delivers a complete, standalone solution for backup and recovery on Linux, with an emphasis on reliability, modularity, and practical deployment across varied environments.

1 Introduction

1.1 Background

In Linux-based environments, data loss can occur due to hardware failures, system crashes, or accidental deletions. While various backup tools exist, they often fall short in providing a complete solution. Some lack essential features like data verification and integrity checks, while others require complex manual configurations that are not user-friendly. This highlights the need for a reliable, automated, and secure backup system tailored specifically for Linux systems.

1.2 Problem Statement

To develop a backup and recovery system for Linux that supports incremental and differential backups, automated scheduling, fast and reliable recovery processes, and backup verification and integrity checks.

1.3 Project Overview

ResilioZ is a backup and recovery system developed to address the limitations of existing Linux-based solutions. It supports full, incremental, and differential backups using content-defined chunking (FastCDC) and deduplication to minimize storage usage. To ensure data reliability and security, metadata is encrypted using AES-256 encryption, and file integrity is validated using SHA256 checksums.

ResilioZ is compatible with multiple storage backends, including local filesystems, NFS-mounted directories, and remote servers accessed over SSH/SFTP. The system provides both a command-line interface (CLI) for power users and a graphical user interface (GUI) built using Qt for better accessibility. Additionally, it features an automated scheduling system implemented as a background daemon, allowing users to define backup intervals with minimal configuration.

1.4 Scope

This project focuses on implementing a complete file-level backup and recovery system tailored for Linux environments. The system is designed to operate at the level of individual files and directories, rather than performing full disk imaging or partition cloning. It ensures that only the necessary data is backed up, thereby optimizing storage space and improving backup and restore performance.

The scope includes:

- **Support for multiple backup types:** Full, incremental, and differential backups, based on change detection using file size and modification time.
- **Repository management:** Creation, selection, deletion, and secure access to local, network-mounted (NFS), and remote (SSH/SFTP) storage backends.
- **Chunking and deduplication:** Use of content-defined chunking (FastCDC) to eliminate redundant data across backups.
- **Compression and encryption:** Zstandard (ZSTD) for fast, high-ratio compression, and AES-256 for secure encryption of backup metadata.

- **Backup verification:** Integrity checks at file level using SHA256 checksum, with metadata verification at restore time.
- **Automated scheduling:** Integration with a background daemon that runs scheduled backups using cron-like syntax via the `libcron` library.
- **User interface support:** Both a CLI and a Qt-based GUI, enabling users to operate the system from either terminal or desktop environments.

2 System Requirements

2.1 Functional Requirements

- The system shall support **full, incremental, and differential backup strategies** using change detection based on file metadata such as size and modification time.
- The system shall enable **accurate data restoration with integrity verification**, using file-level SHA256 checksums and generating detailed reports of any mismatches detected during restoration.
- The system shall provide functionality to **create, manage, and delete repositories** across multiple storage back-ends (local and remote storages) through a **unified interface**.
- The system shall support **automated backup scheduling** via a client-server scheduler component.
- The scheduler shall support **creating, viewing, and removing scheduled backup tasks**, along with appropriate **logging and error reporting** for operational transparency.

2.2 Non-Functional Requirements

- The system shall be **platform independent across major Linux distributions** by leveraging standard C++ libraries and cross-platform development tools.
- The system architecture shall be **modular**, abstracting storage-specific functionality through **unified interfaces** to ensure consistent behavior across filesystems and network protocols.
- The user interfaces shall remain **responsive**, completing standard operations such as menu navigation, backup initiation, restoration, and repository management. This shall be achieved through optimized change detection, memory-efficient file processing, and low-latency storage interactions.

3 System Architecture and Design

3.1 UML Diagrams

3.1.1 Class Diagram

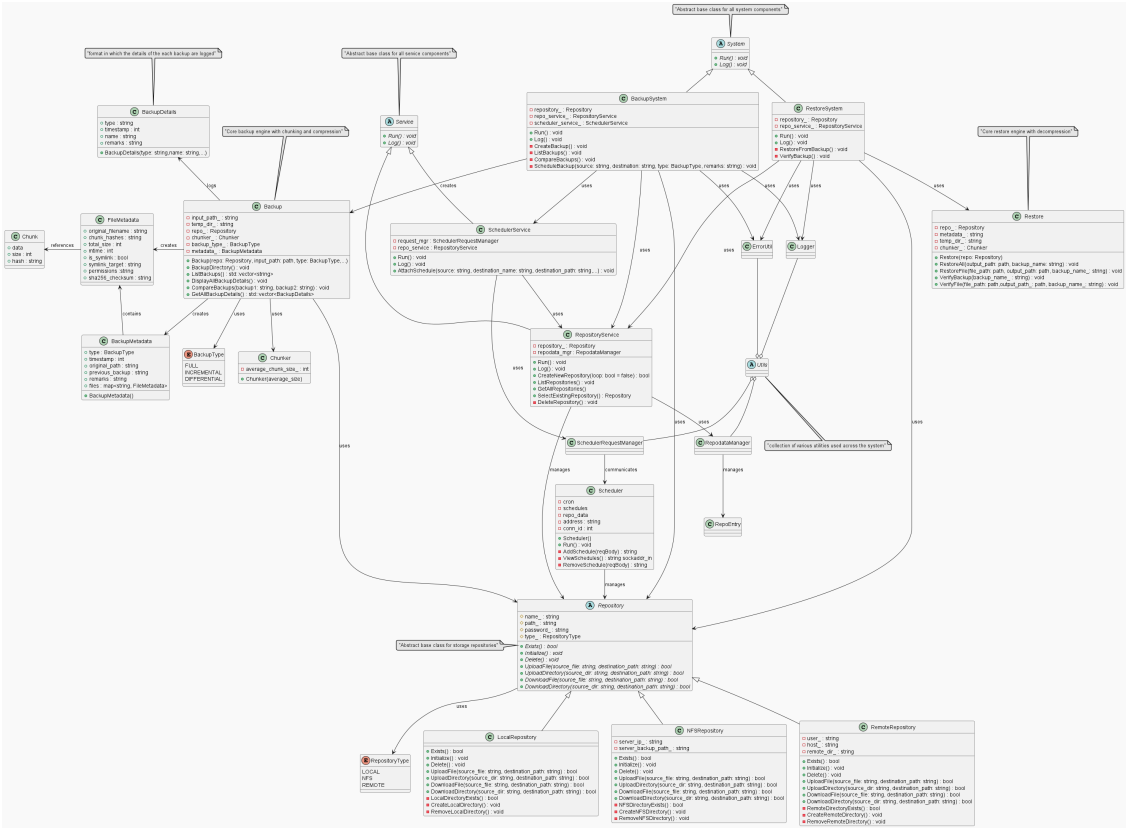


Figure 3.1: Class Diagram showing core components and their relationships

3.1.2 Sequence Diagram

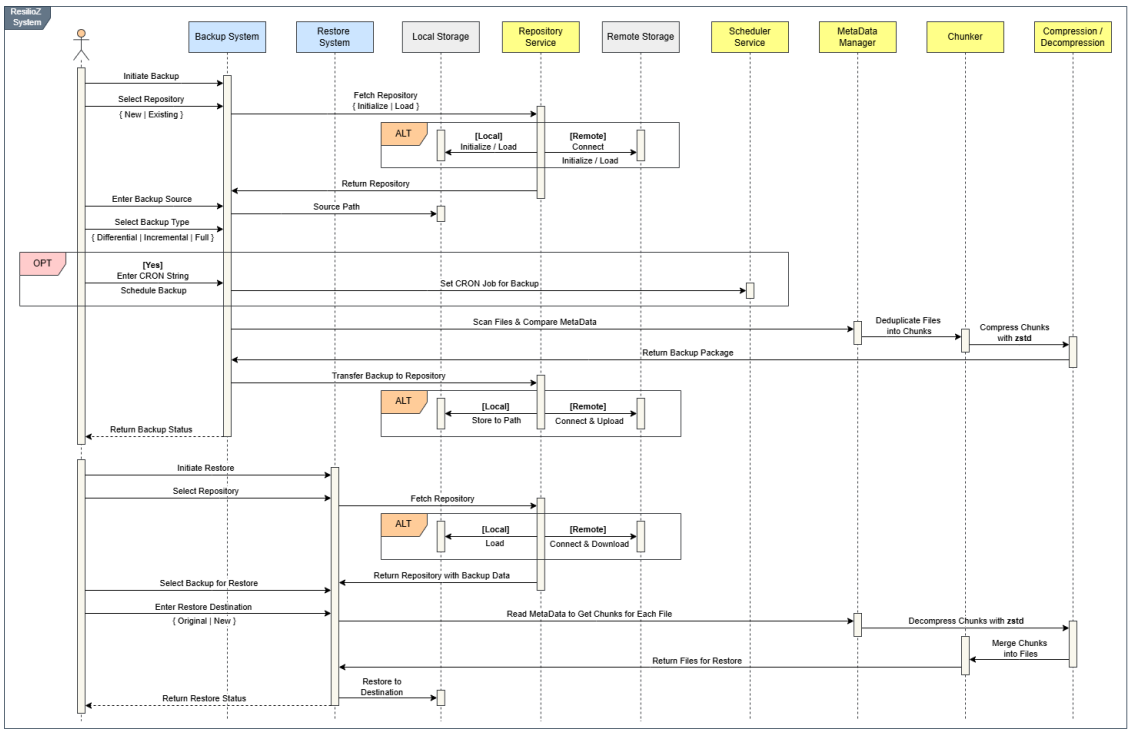


Figure 3.2: Sequence Diagram for Backup and Restore Operations

3.2 System Workflow

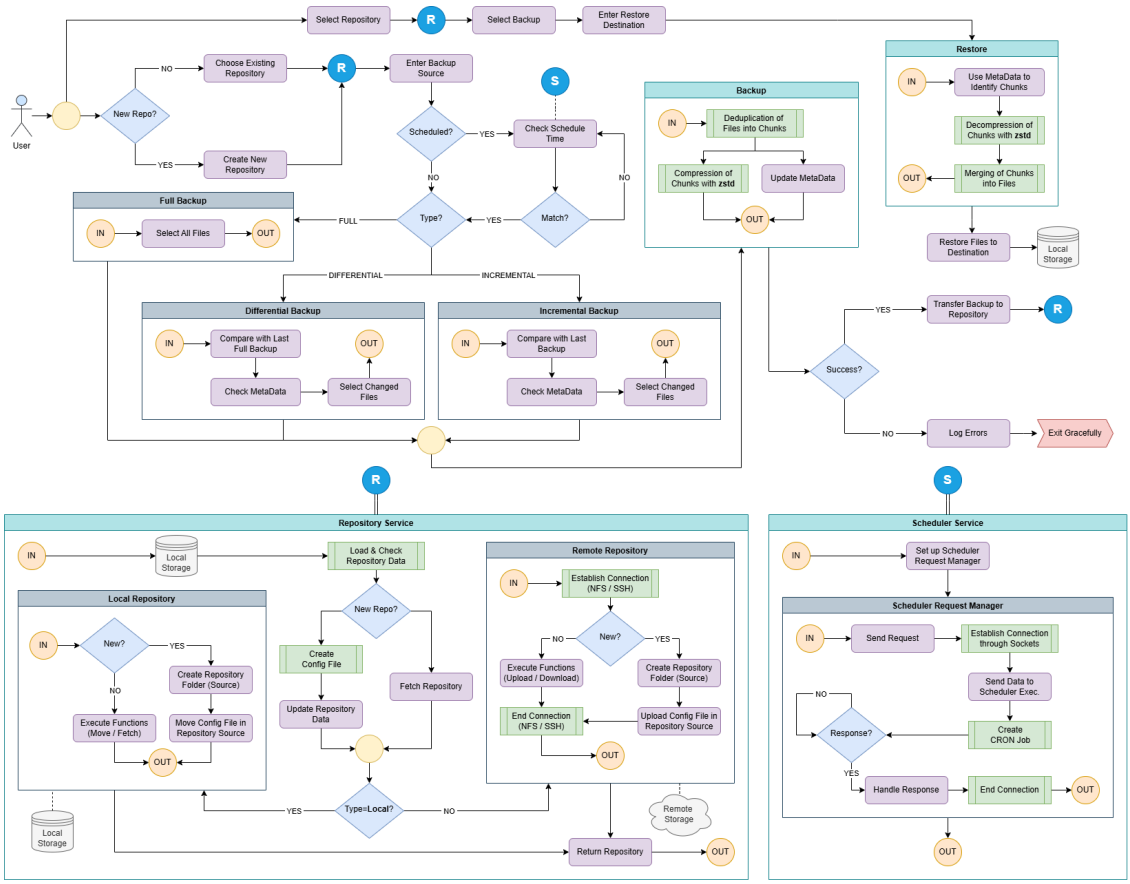


Figure 3.3: A flow-chart of the system

The workflow of the Linux Backup and Recovery System is designed to provide seamless, automated, and reliable data protection through coordinated module interactions, as depicted in the sequence diagram. When a user initiates a backup—either manually or via the automated scheduler, the system determines the backup type (full, incremental, or differential) and scans the filesystem, comparing the current file metadata with prior snapshots to identify changes. Changed files are then passed to the deduplication and compression engine, which eliminates redundancy and compresses the data. The system applies encryption to the metadata to further secure the backup contents. The finalized backup package is stored locally or it is uploaded to remote storage (via NFS or SSH/SFTP). The metadata manager updates the repository with the new backup details, while logs are written to track actions and potential issues. The recovery system uses metadata to identify the necessary files and retrieves the corresponding chunks. These are decompressed and decrypted before being restored to the desired destination, maintaining original metadata and file integrity. Repository interaction is abstracted, allowing seamless transitions across local, NFS, or remote storage environments. This modular, fault-tolerant workflow ensures secure, efficient, and scalable backup and recovery operations.

4 Core Implementation

4.1 Backup System

The Backup System in ResilioZ represents the core engine that implements sophisticated data protection capabilities through advanced chunking, deduplication, compression, and encryption technologies. This system provides advanced backup functionality with support for multiple backup types, intelligent change detection, and comprehensive metadata management to ensure reliable data protection across various storage environments.

4.1.1 Purpose

The Backup System addresses critical data protection challenges in backup environments through comprehensive backup capabilities that ensure data integrity, storage efficiency, and operational reliability. The system eliminates data redundancy by performing intelligent deduplication to optimize storage utilization and reduce costs. The backup system keeps detailed records and allows backups to be verified, making it easy to track past actions. It enables quick and reliable data recovery, and supports different backup types — Full, Incremental, and Differential — to suit various operational needs. The system also optimizes resource utilization through advanced compression algorithms and intelligent chunking techniques that minimize storage requirements while maintaining high performance.

4.1.2 System Architecture

Core Components Integration: The Backup System implements a sophisticated architecture that integrates multiple specialized components to provide comprehensive backup functionality. The system consists of the Backup class which serves as the primary orchestrator and the Chunker class that implements content-defined chunking and the Repository system for storage abstraction across different storage types. The architecture follows a modular design where each component has specific responsibilities while maintaining clean interfaces for integration. The Backup class manages the overall backup process including metadata handling, file processing, and repository coordination. The Chunker class handles the complex task of breaking files into content-defined chunks for deduplication. The system integrates seamlessly with the Repository system to support multiple storage backends including local filesystems, NFS network storage, and remote SSH/SFTP storage. This integration enables flexible backup deployment across various infrastructure configurations while maintaining consistent backup functionality and data integrity.

Data Flow Architecture: The backup data flow follows a pipeline that begins with file system scanning and metadata collection, progresses through content-defined chunking and deduplication, and concludes with compression, encryption and storage in the repository system. This pipeline ensures optimal storage efficiency while maintaining data integrity and security throughout the backup process. The system implements intelligent heuristic change detection that compares file metadata using size and modification time to determine which files require backup processing. This approach minimizes backup time and storage requirements by avoiding unnecessary processing of unchanged files while ensuring comprehensive data protection.

4.1.3 Core Functionalities

Backup Type Management: The Backup System supports three fundamental backup types that address different operational requirements and storage optimization strategies. Full backups create complete snapshots of all data, providing comprehensive data protection and enabling complete system recovery. These backups serve as baseline references for incremental and differential backup operations while ensuring that all data is protected regardless of previous backup state. Incremental backups build upon the most recent backup of any type by storing only files that have changed since the last backup operation. The system implements sophisticated change detection that compares file metadata including file size and modification time, to determine which files require backup processing. Differential backups store changes since the last full backup, providing a middle ground between full and incremental approaches. The system identifies the most recent full backup using the `GetLatestFullBackup()` function and uses it as the reference point for determining which files have changed.

File Processing and Metadata Management: The Backup System implements comprehensive file processing that handles various file types which include regular files, symbolic links, and directories with appropriate processing strategies for each type. Regular files undergo content-defined chunking for deduplication and compression, while symbolic links are processed by storing their target paths and metadata without content processing. The system maintains detailed metadata for each file using the original filename, file permissions in octal format, modification time, total size, and SHA256 checksum for integrity verification. This metadata is stored in encrypted JSON format within the repository, ensuring that file information remains secure and accessible for restore operations. The system uses the repository password for metadata encryption, ensuring that backup information remains secure even if the repository storage is compromised.

Content-Defined Chunking and Deduplication: The system implements content-defined chunking using the FastCDC algorithm to enable effective deduplication across multiple backups and files. This approach breaks files into variable-sized chunks based on content characteristics rather than fixed boundaries, ensuring that identical content produces identical chunks regardless of file context. The chunking system uses a 64-byte sliding window with gear hashing to identify optimal chunk boundaries that maximize deduplication efficiency while maintaining reasonable chunk sizes. The system implements a three-region approach with minimum, normal and maximum size constraints to ensure chunk sizes remain within optimal ranges for storage and processing efficiency. Each chunk receives a SHA256 hash for identification and deduplication purposes, enabling the system to store identical chunks only once across multiple backups. This deduplication capability provides significant storage savings, particularly for incremental and differential backups where many chunks remain unchanged between backup operations.

Compression and Storage Optimization: The Backup System implements ZSTD compression for optimal balance between compression ratio and processing speed. Each chunk is compressed individually before storage, with the original size stored as a prefix to enable proper decompression during restore operations. The compression system calculates the maximum compressed size and creates buffers that include both the original size and compressed data. The system implements intelligent chunk storage with hash-based directory organization to prevent filesystem performance degradation when storing large numbers of small files.

Chunks are organized in subdirectories based on the first two characters of their hash, distributing storage load across multiple directories.

Encryption and Security: The Backup System implements comprehensive encryption for metadata using the repository password through the `EncryptionUtil` system. All backup metadata including file information, chunk references, and backup configuration is encrypted before storage, ensuring that backup information remains secure even if the repository storage is compromised. The system uses AES-256 encryption for metadata protection, providing for sensitive backup information. The encryption process is transparent to users while ensuring that backup metadata cannot be accessed without the repository password. The system implements SHA256 checksums at the file level, storing them in the metadata to ensure integrity verification during restore operations. Additionally, each chunk's filename is derived from its SHA256 hash, reinforcing consistency and traceability throughout the backup and restore process.

4.1.4 Metadata Format Structure

The backup system uses a JSON-based metadata format that is encrypted using AES-256 encryption with the repository password. The metadata structure consists of two main components: backup-level metadata and file-level metadata, providing comprehensive information about backup operations and file characteristics.

Backup-Level Metadata Structure: The backup-level metadata contains essential information about the backup operation including the backup type as an integer enumeration, the precise timestamp when the backup was created, the name of the previous backup for incremental and differential backups, and optional user remarks for documentation purposes. The metadata structure follows this JSON format:

```
{
    "type": 0,                    // 0=Full, 1=Incremental, 2=Differential
    "timestamp": 1703123456,      // Unix timestamp in seconds
    "previous_backup": "",        // Name of previous backup (empty for full)
    "remarks": "",   // Optional user comments
    "files": {}                   // File metadata dictionary
}
```

The backup type field uses integer enumeration where 0 represents a full backup, 1 represents an incremental backup, and 2 represents a differential backup. The timestamp field stores the precise creation time as a Unix timestamp in seconds, enabling accurate temporal tracking of backup operations. The `previous_backup` field contains the filename of the backup that serves as the reference point for incremental and differential operations, while the remarks field allows users to add descriptive comments about the backup purpose or context.

File-Level Metadata Structure: Each file entry in the metadata contains comprehensive information about the file including the original filename, an ordered list of chunk hashes that represent the file's content, the total file size in bytes, the file modification time as a Unix timestamp, file permissions in octal format, and a SHA256 checksum of the entire file for integrity verification. The file metadata structure follows this format:

```
‘path/to/file : {
    "original_filename": "document.txt",
    "chunk_hashes": [
                "a1b2c3d4e5f6...",
                "f6e5d4c3b2a1...",
                "1234567890ab..."
            ],
    "total_size": 1048576,
    "mtime": 1703123456,
    "is_symlink": false,
    "permissions": "0644",
    "sha256_checksum": "a1b2c3d4e5f6...",
    "symlink_target": "/path/to/target"  // Only for symlinks
}
```

The `original_filename` field stores the base name of the file without its full path, enabling proper restoration with the correct file naming. The `chunk_hashes` array contains an ordered list of SHA256 hashes corresponding to the chunks that make up the file, allowing accurate reconstruction during restore operations. The `total_size` field stores the uncompressed size of the file in bytes, while the `mtime` field preserves the file's modification timestamp to ensure accurate restoration of file metadata.

Symlink Handling: The metadata handler handles symbolic links differenetly through the `is_symlink` flag and `symlink_target` field. When a symbolic link is encountered, the system stores the target path in the `symlink_target` field and sets `is_symlink` to true. For symlinks, the `chunk_hashes` array remains empty as symbolic links don't contain file content, and so the field `sha256_checksum` is also empty as there is no content to hash. The symlink handling ensures that symbolic links are properly preserved during backup and restore operations, maintaining the file system structure and relationships.

Metadata Encryption and Storage: The complete metadata structure is serialized to JSON format with proper indentation for readability and debugging purposes. The JSON string is then encrypted using AES-256 encryption with the repository password as the encryption key through the `EncryptMetadata()` function. The encrypted metadata is stored as a binary file in the repository with the backup timestamp as the filename. The encryption ensures that backup metadata remains secure even if the repository storage is compromised, protecting sensitive information about file structures, timestamps, and file relationships. The system automatically handles decryption when loading metadata for restore operations or backup comparisons, providing transparent access to backup information while maintaining security. The metadata files are stored in the repository under the `backup/` directory with filenames corresponding to the backup timestamp in the format `YYYYMMDD_HHMMSS`. This naming convention ensures unique identification of each backup while providing chronological ordering for backup management operations.

4.1.5 Repository Integration

Storage Abstraction: The Backup System integrates seamlessly with the Repository service to provide storage abstraction across multiple storage backends. This integration enables the system to work with lo-

cal filesystems, NFS network storage, and remote SSH/SFTP storage without requiring changes to the core backup logic. The repository integration provides consistent interfaces for file upload, download, and metadata operations regardless of the underlying storage technology. This abstraction enables flexible deployment across various infrastructure configurations while maintaining consistencyy.

Multi-Repository Support: The system supports all three repository types with appropriate optimizations for each storage technology. Local repositories provide the highest performance for backup and restore operations, while NFS repositories enable network-based storage with reasonable performance characteristics. Remote repositories use SSH/SFTP for secure remote storage, providing off-site backup capabilities with encryption and authentication.

4.1.6 Error Handling and Reliability

Comprehensive Error Management: The Backup System implements robust error handling that addresses various failure scenarios, including file access issues, compression failures, and metadata corruption. The system uses the `ErrorUtil` module for consistent error reporting and logging throughout all backup operations. It provides detailed error messages to help users and administrators understand the root cause of failures and take appropriate corrective actions.

Data Integrity Verification: The system ensures data integrity using SHA256 checksums. File-level checksums are used to verify the integrity of files. Integrity checks are performed during backup verifications and restore operations to detect any data corruption, enabling early identification of storage or transmission issues. All integrity failures are logged for future investigation and are appropriately reported to the user.

4.1.7 Performance Optimizations

Processing Efficiency: The system implements several processing-level optimizations including streaming operations, intelligent change detection, and high-performance compression. The streaming-based architecture enables processing of files that are larger than available memory, without degrading performance. The change detection logic compares file metadata such as size, modification time, and hash values to avoid reprocessing unchanged files, significantly improving backup efficiency. Zstandard (ZSTD) compression is used to provide a strong balance between speed and compression ratio.

Storage Optimization: Storage efficiency is achieved using content-defined chunking, deduplication, and intelligent chunk distribution. The FastCDC algorithm produces variable-length chunks that enhance deduplication while maintaining reasonable chunk sizes. These chunks are compressed using ZSTD, and then organized into subdirectories based on hash prefixes, which avoids filesystem performance issues when storing a large number of small files.

4.1.8 Security Considerations

Data Encryption: The system applies AES-256 encryption to all metadata using repository-specific passwords. This includes encryption of file information, chunk references, and backup configuration. The encryption process is integrated into the backup and restore workflows, ensuring that metadata remains pro-

tected without requiring additional steps from the user. Encryption keys are derived from user-provided passwords, preventing unauthorized access to sensitive backup data.

Access Control: Access to backup data is protected through repository-level authentication and file system permissions. During backup operations, the system respects existing permissions and ensures that backup data is not exposed to unauthorized users. Access to repository operations also requires password verification, adding another layer of security.

Audit Trails: Comprehensive audit trails are maintained through detailed logs of all backup-related activities such as file traversal, and repository interactions. These logs assist in troubleshooting and operation tracking.

4.2 Restore System

The Restore System in ResilioZ represents the critical data recovery engine that implements file reconstruction, integrity verification, and metadata restoration capabilities. This system provides proper recovery functionality with support for streaming operations, comprehensive error handling, and multi-repository compatibility to ensure reliable data restoration across various storage environments and backup types.

4.2.1 Purpose

The Restore System addresses critical data recovery challenges in backup environments through comprehensive restoration capabilities that ensure data accessibility, integrity verification, and operational reliability. The system eliminates data loss risks by implementing recovery mechanisms that reconstruct files from compressed and chunked backup data while providing intelligent integrity verification to ensure data accuracy. The system also optimizes resource utilization through streaming reconstruction algorithms and intelligent memory management that minimize storage and processing requirements while maintaining high performance.

4.2.2 System Architecture

Core Components Integration: The Restore System implements an architecture that integrates multiple specialized components to provide comprehensive data recovery functionality. The system consists of the Restore class which serves as the primary reconstruction engine, the `RestoreSystem` class that provides high-level user interface management and the Repository system for storage abstraction across different storage types. The architecture follows a modular design where each component has specific responsibilities while maintaining clean interfaces for integration. The Restore class manages the core restoration process including metadata loading, chunk reconstruction, and integrity verification. The `RestoreSystem` class handles user interaction and repository coordination. The system integrates seamlessly with the Repository system to support multiple storage backends including local filesystems, NFS network storage, and remote SSH/SFTP storage. This integration enables flexible recovery deployment across various infrastructure configurations while maintaining consistent functionality and performance across different storage environments.

4.2.3 Core Functionalities

Metadata Loading and Decryption: The Restore System implements sophisticated metadata loading and decryption capabilities that enable access to backup information regardless of the storage location or encryption status. The system downloads encrypted metadata from the repository storage and uses the repository password to decrypt the backup information using AES-256 decryption algorithms. The metadata loading process begins with file existence validation to ensure the requested backup is available in the repository storage. The system then performs binary file reading operations to load the encrypted metadata into memory, followed by AES-256 decryption using the repository password as the encryption key. The decrypted metadata undergoes JSON parsing to extract backup information including file structures, chunk references, and metadata attributes. The system implements comprehensive error handling for corrupted or inaccessible backup files through graceful degradation mechanisms that allow restoration to continue even if individual backup files are damaged.

File Reconstruction Engine: The Restore System implements sophisticated file reconstruction capabilities that enable accurate restoration of files from compressed, chunked backup data. The system uses the chunk hashes stored in metadata to locate and download the necessary compressed chunks from the repository storage, followed by decompression and sequential combination to reconstruct the original file content. The file reconstruction process implements streaming operations that enable restoration of files larger than available system memory while maintaining optimal performance characteristics. The system downloads chunks on-demand as they are needed for file reconstruction, minimizing memory usage and enabling restoration of large filesystems without resource exhaustion. The reconstruction engine handles different file types with appropriate processing strategies including regular files that undergo complete chunk reconstruction, symbolic links that are restored by creating the appropriate link relationships, and directories that are recreated with proper hierarchical structure. The system implements intelligent path reconstruction that preserves the original directory hierarchy while adapting to the chosen restore location.

Chunk Processing and Decompression: The Restore System implements sophisticated chunk processing and decompression capabilities that enable accurate data reconstruction from compressed backup storage. The system uses the chunk hashes stored in metadata to locate and download compressed chunks from the repository storage, followed by ZSTD decompression to restore the original data content. The chunk processing system implements streaming operations that process chunks sequentially without loading entire files into memory. Each downloaded chunk undergoes ZSTD decompression using the `ZSTD_decompress` function with proper error handling for decompression failures. The system calculates the original chunk size from the compressed data prefix and allocates appropriate buffers for decompression operations. The chunk processing includes comprehensive error handling for download failures, decompression errors, and data corruption scenarios.

Integrity Verification System: The Restore System implements comprehensive integrity verification that ensures restored files match their original content exactly. The system calculates SHA256 checksums for restored files using the OpenSSL library and compares them against the checksums stored in the backup metadata to verify data integrity and accuracy. The integrity verification process begins after successful file reconstruction and includes both file-level and chunk-level verification to ensure complete data integrity.

The system implements efficient checksum calculation using buffered reading operations that process files to provide accurate hash computation. The verification system provides detailed reporting of any discrepancies found during the restore operation, enabling identification of potential data corruption or restoration issues. The system implements graceful handling of integrity failures that allows the restore operation to continue while providing comprehensive reporting of any issues encountered. This approach ensures that users receive complete information about the restore operation status and can take appropriate action for any files that require attention.

Path Reconstruction and Directory Management: The Restore System implements intelligent path reconstruction that preserves the original directory hierarchy while adapting to the chosen restore location. The system analyzes the original file paths from backup metadata and reconstructs the complete directory structure within the specified restore destination. The path reconstruction process handles both original location restoration and custom destination paths with appropriate directory creation and path modification. For original location restoration, the system uses the root directory as the base path and reconstructs the complete file tree. For custom locations, the system creates the necessary directory structure within the specified destination path while maintaining the original hierarchical relationships. The directory management system implements automatic directory creation that ensures all necessary parent directories exist before file restoration begins. The system handles path modifications that remove leading slashes and adapt absolute paths to relative paths within the restore destination. This approach ensures that restored files maintain their original organizational structure while adapting to the chosen restore location.

Metadata Preservation and Restoration: The Restore System implements comprehensive metadata preservation that ensures restored files maintain their original characteristics including permissions, timestamps, and symbolic link relationships. The system extracts metadata information from backup storage and applies it to restored files to maintain complete fidelity with the original data. The metadata preservation process includes file permission restoration that converts stored octal permission strings to filesystem permission objects using the `fs::permissions` function. The system handles permission conversion by parsing the octal string and mapping individual permission bits to the appropriate filesystem permission flags. The system implements graceful handling of permission restoration failures that logs warnings without interrupting the restore process. The timestamp restoration process uses the `fs::last_write_time` function to restore the original file modification timestamps from backup metadata. The system converts stored Unix timestamps to filesystem time types and applies them to restored files to maintain temporal accuracy. The symbolic link restoration process creates appropriate symbolic link relationships with correct target paths for both file and directory symbolic links.

4.2.4 Error Handling and Reliability

Comprehensive Error Management: The Restore System implements robust error handling that addresses various failure scenarios including repository connectivity issues, metadata corruption, chunk download failures, and decompression errors. The system uses the `ErrorUtil` system for consistent error reporting and logging throughout all restore operations. The error handling system provides detailed error messages that help users and administrators understand the root cause of failures and take appropriate corrective actions. The system implements graceful degradation to continue operation despite individual file or

chunk failures, ensuring that partial restorations can be completed successfully. This state tracking includes information about successfully restored files, failed operations, and integrity verification results.

Result Tracking and Reporting: The Restore System implements a result tracking that maintains comprehensive information about restoration operation outcomes. The system tracks successful restorations, failed operations, and integrity verification results to provide detailed operation reporting and enable appropriate corrective actions. The result tracking system maintains three primary result categories including successful files that were restored without issues, failed files that encountered errors during restoration, and integrity failures that passed restoration but failed integrity verification. This categorization enables detailed operation analysis and appropriate corrective actions. The reporting system implements intelligent result analysis that provides appropriate log levels and user feedback based on operation outcomes. The system provides detailed summaries including total file counts, success rates, and specific failure information to enable comprehensive operation analysis and decision-making.

4.2.5 Security Considerations

Data Integrity Verification: The Restore System implements comprehensive data integrity verification that ensures restored files match their original content exactly. The system uses SHA256 checksums to verify file integrity and provides detailed reporting of any discrepancies found. The integrity verification process includes both file-level and chunk-level verification to ensure complete data integrity. The system implements robust error handling for integrity failures that provides clear reporting and enables appropriate corrective actions.

Secure Metadata Handling: The Restore System implements secure metadata handling that ensures backup information remains protected during restore operations. The system uses the repository password for metadata decryption and implements secure cleanup of decrypted metadata after use. The system implements appropriate access control through repository authentication and file system permissions. The system respects file system permissions during restore operations and maintains appropriate access controls for restored data.

Audit Trails and Logging: The Restore System provides comprehensive audit trails through detailed logging of all restore operations including file processing, chunk reconstruction, and integrity verification. The logging system records operation details and error conditions for security monitoring and operational analysis.

4.3 Repository Service

The Repository Service in ResilioZ functions as the central coordination layer for managing repositories across the application. It provides a high-level interface for operations such as creating, selecting, and deleting repositories, abstracting the underlying storage details from the systems that consume them, such as the Backup and Restore modules. By isolating these management tasks into a dedicated service, the application ensures better modularity, separation of concerns, and consistent repository handling across different components. The service handles local, NFS, and remote repositories with enforced password verification, ensuring secure repository management.

4.3.1 Service Reponsibilities

- **Creating new repositories** based on user input, including initialization of configuration files and folder structures.
- **Selecting existing repositories**, validating their presence and integrity before exposing them to other systems.
- **Deleting repositories** safely, ensuring that system-wide references and metadata are also cleaned up.
- **Providing validated repository instances** to core components like Backup and Restore, abstracting internal details.

To support these functionalities efficiently, the service operates in coordination with the utility class `RepodataManager`, which manages a central tracking file named `repodata.json`. This file stores summary-level metadata for all registered repositories, allowing for faster lookups and password verification without the need to access each repository's internal configuration individually. This clear separation of responsibilities enables core systems to function independently of repository management logic, contributing to a more modular system architecture and simplifying long-term maintenance.

4.3.2 Repository

A repository serves as the foundational storage layer that manages and organizes backup data across different storage mediums. It provides a unified interface for storing, retrieving, and managing backup files regardless of whether they are stored locally, or on remote servers. All the repositories maintain their own internal structure with dedicated subdirectories for different types of backup data, ensuring organized storage and efficient data management.

4.3.3 Internal Structure

Each repository (stored in a folder named after it) follows this fixed structure:

- **File** `config.json`: This file stores important settings related to the specific repository, such as the repository type (local or remote), storage path, and the `password_hash` used to verify user access.
- **Folder** `backups/`: Each backup is accompanied by a metadata file in JSON format, which includes a list of backed-up files along with their original filenames, sizes, SHA256 checksums, associated chunk hashes, file permissions, and modification timestamps. This metadata is essential during restore operations to ensure accurate reconstruction and verification of the original data.
- **Folder** `chunks/`: The actual file data is split into small pieces called chunks. These are stored in subfolders named by the first two characters of the chunk's hash, ranging from `00` to `ff`. This helps keep the folder clean and improves performance, especially when there are thousands of chunks.

4.3.4 Central Repository Record

As stated earlier, to make repository access faster and more efficient, the system maintains a separate file named `repodata.json`, which stores an array of JSON objects, each describing details of every repository

(similar to what's inside each `config.json`). This file is stored centrally and is used for quick lookups during operations like password verification, without having to load each individual `config.json`.

4.3.5 Repository Types

Local Repository: The local repository in ResilioZ is designed to provide fast backup and restore operations by storing all backup data directly on the same machine or a locally attached storage device. It leverages direct filesystem access for high performance and minimal latency, making it ideal for standalone systems and development environments. For comprehensive protection against hardware failure or disasters, they are best complemented with networked or remote repositories, ensuring both speed and resilience in the backup strategy.

Purpose: Local repositories are used when users need to store backup data on the same machine or on directly attached storage devices. They provide the fastest access times since data is stored locally without network overhead, making them ideal for:

- Frequent backup operations
- Quick restore processes
- Scenarios with unreliable or unavailable network connectivity

Input: To create a local repository, the user must provide:

- Repository Name
- Local Path
- Repository Password (used to encrypt sensitive metadata)

The system validates the inputs, checks for existing repositories at the specified location, and creates a structured directory hierarchy containing organized storage areas for backup data, chunks, and configuration files.

NFS Repository: An NFS repository is a storage container that manages backup data on a remote NFS server accessible over the network. It creates a structured directory hierarchy on the NFS share, similar to local repositories, but stores all data on a remote server preferably connected over private networks or LANs.

NFS: NFS (Network File System) is a distributed file system protocol that allows users to access files over a network as if they were stored locally. It operates at the application layer and uses Remote Procedure Calls (RPC) to perform file operations. NFS supports seamless file sharing across different systems and platforms without requiring specialized configuration at the application level.

Purpose: The purpose of NFS-based storage in ResilioZ is to provide a shared, network-accessible location for storing backup data. NFS repositories are ideal for:

- Centralized backup across systems
- Efficient storage and easier management
- Scenarios with high LAN performance

Input: To create an NFS repository, the user must provide:

- NFS Path in the format `hostname:/path`
- Repository Name
- Repository Password

The system validates the NFS path format using regex to ensure it follows the correct `hostname:/absolute/path` structure. The NFS server must also be properly configured and accessible from the client machine.

Remote Repository: A remote repository is a storage container that manages backup data on a remote server accessible via SSH/SFTP. It creates a structured directory hierarchy on the remote machine, similar to local and NFS repositories, and stores all data securely over private or public networks.

SSH: SSH (Secure Shell) is a cryptographic protocol that enables secure communication over unsecured networks. It supports encrypted authentication and data transfer, allowing users to access and manage remote systems securely. SSH can use various authentication methods, such as passwords, public keys, and certificates. Our system relies on public key based authentication for smooth repository connections.

SFTP: SFTP (SSH File Transfer Protocol) is a secure file transfer protocol that operates over SSH. It encrypts all transferred data, including credentials and file contents. SFTP supports a wide range of file operations and works across different operating systems, making it suitable for secure and reliable backup operations.

Purpose: Remote repositories serve multiple critical use cases:

- Off-site backup storage for disaster recovery
- Secure cloud-based backups
- Encrypted data transfers over public networks

Input: To create a remote repository, the user must provide:

- SFTP Path in the format `user@hostname:/path`
- Repository Name
- Repository Password

The system validates the SFTP path using regex to ensure it conforms to the `user@host:/absolute/path` structure. SSH key-based authentication must be properly set up between the client and server to enable secure access.

4.3.6 Core Functionalities

File Upload Operations: The system's file upload functionality is adapted across repository types to leverage the most efficient transfer method for each backend. For Local repositories, it employs direct, high-performance filesystem operations using standard library methods of `std::filesystem`.

For NFS repositories, the system abstracts network operations using the `libnfs` library to establish connections, creating remote file handles, and transferring data buffers to optimize for LAN speeds.

For secure Remote transfers, the system utilizes the `libssh` library to establish an SFTP session, authenticating via public key and reading or writing data in encrypted chunks. Each implementation ensures proper session and resource cleanup tailored to its respective library.

Directory Upload Operations: Uploading directories is handled through recursive traversal, with each repository type implementing the logic best suited for its environment. The Local repository performs direct recursion using `std::filesystem::recursive_directory_iterator`, calculating relative paths to precisely replicate the source structure in the destination.

Both NFS and Remote repositories employ a custom recursive lambda function for traversal; the NFS implementation creates remote directories using `nfs_mkdir`, while the Remote SFTP version uses `sftp_mkdir`.

File Download Operations: File download operations mirror the upload logic, prioritizing performance and security appropriate for each storage backend. A Local download is a direct and fast filesystem copy using `fs::copy_file`, with validation to ensure the source file exists and is regular.

For NFS downloads, the system establishes a connection and opens the remote file in read-only mode using `nfs_open`, reading contents into efficient 1MB buffer with `nfs_read`, then writing to a local stream.

Remote downloads follow a similar pattern over a secure channel, using `libssh` to initialize an SFTP session, opening the file with `sftp_open`, and reading encrypted data in 16KB buffer via `sftp_read`, which are then written to local storage, ensuring secure and efficient retrieval.

Directory Download Operations: The system reconstructs complete directory hierarchies from a repository to a local destination via remote traversal. For Local repositories, this is a straightforward recursive copy operation.

For NFS repositories, the system iterates through directories using `nfs_opendir` and `nfs_readdir`, checking each entry's type with `nfs_stat64` to differentiate between files and directories.

Similarly, the Remote SFTP implementation uses `sftp_opendir` and `sftp_readdir` to traverse and inspects `sftp_attributes` for type detection. In both cases, the system recursively replicates the directory structure on the local filesystem and invokes the appropriate single-file download method for each file, ensuring accurate reconstruction of the directory tree.

4.4 Scheduler System

The Scheduler System in ResilioZ is a sophisticated automated backup scheduling solution that operates as a client-server architecture. It provides users with the ability to create, manage, and execute time-based backup operations without manual intervention, ensuring consistent and reliable data protection through automated processes.

4.4.1 Purpose

The Scheduler System addresses critical operational challenges in general backup environments through comprehensive automation. The system eliminates the need for manual intervention by ensuring backups occur at precisely scheduled intervals regardless of human availability, significantly reducing human error and missed backup windows while enabling continuous data protection without human oversight. It provides verifiable backup execution logs and schedules.

4.4.2 System Architecture

Client-Server Architecture: The Scheduler System implements a distributed architecture with two main components that work together to provide seamless scheduling capabilities. The client component, integrated

within the main application executable, consists of the `SchedulerService` which provides the user interface for schedule management, and the `SchedulerRequestManager` which handles the network communication layer between the executables. The server component operates as a standalone executable called scheduler that runs as a background daemon process utilising `systemd`, containing the Scheduler class which serves as the core scheduling engine and server.

Communication and Connectivity: The system uses TCP socket communication for client-server interaction, providing reliable and efficient communication between components. The server is exposed on `localhost` and port `55055`, and utilises TCP/IP protocol for message transfer and communication. The client configuration establishes a connection to the scheduler executable through a socket, exchanges JSON payloads for requests and responses, and implements immediate connection failure handling when the server is unavailable.The communication flow follows a structured pattern where the client sends JSON requests through TCP sockets to the server, which processes the requests and executes backup operations accordingly. The server then logs results and sends the responses back to the client, which updates the user interface based on the received information.

4.4.3 Core Functionalities

Schedule Creation and Management: The scheduler module provides schedule creation with comprehensive input validation for cron expressions, repository parameters, and backup types. It dynamically creates the appropriate repository objects for Local, NFS, or Remote storage which are then used at the time of execution of backup. The system stores schedule metadata and repository information in memory-mapped data structures and registers schedules with the libcron library for execution, generating unique schedule identifiers for effective management. The schedule metadata structure maintains essential information including repository objects, cron expression for timing, backup source path, user remarks, and unique schedule identifier. This structured approach ensures all necessary information is preserved for reliable schedule execution. The system supports comprehensive error handling with validation feedback for all operations both in the network and application contexts as well as cleanup of dynamically allocated data on deletion of a schedule.

Schedule Execution Engine: The scheduler leverages the `libcron` C++ library for cron expression parsing and execution, utilising cron strings to denote schedules and determining next execution times based on the same. The system automatically triggers backup operations at scheduled intervals, ensuring reliable execution without manual intervention. The backup execution process maintains backup context including source, type, and repository information across executions, retrieving stored repository instances for backup operations. The system handles individual backup failures gracefully, logging errors for failed operations and remains stable and continues to backup non-affected schedules as normal.

Schedule Monitoring and Control: The view schedules functionality enables access of stored schedule metadata from memory and retrieves current repository status and information, converting internal data structures to JSON response format for client consumption. The system ensures data consistency between schedules and repositories while detecting mismatches between metadata and repository data to maintain system integrity. The RemoveSchedule functionality allows for deletion of schedules by unique identifier and performs comprehensive resource cleanup including deallocation of repository instances in memory,

removal of schedules from the libcron execution queue, and cleanup of schedule metadata from storage. The system logs any error or discrepancy found throughout this process to a separate scheduler logs file which can be inspected to validate execution.

4.4.4 CRON Strings

Cron strings are used to indicate specific schedules and follow a six field space separated format. The fields are second (0-59), minute (0-59), hour(0-23), day of month(1-31), month(1-12) and day of week (0-6). Special character asterisk (*) can be used to represent the entire range and question mark (?) is used to ignore day of week / day of month. It also supports lists (1,4,7) or ranges (4-7) as fields as well as steps (m - start /n - end). For cleaner strings mon-sun and jan-dec can be used in week day and month fields respectively.

Example Patterns:

- **0 0 2 * * ?** (2:00 AM daily)
- **0 0 2 * * SUN** (2:00 AM every Sunday)
- **0 0 2 1 * ?** (2:00 AM on the 1st of each month)
- **0 0 * * * ?** (at the top of every hour)
- **0 0 12 * * MON-FRI** (Every Weekday at noon)

4.4.5 Components

Server Implementation (Scheduler): The server component is responsible for managing the scheduler daemon and handling client requests over a network interface. It initializes as a background daemon, binds to TCP port `55055`, and enters a continuous loop to accept and manage client connections. The server uses a multi-threaded architecture, where one thread handles incoming network requests while another is dedicated to executing scheduled tasks via `libcron` Cron objects using periodic `tick()` function calls.

The server includes robust features such as socket creation with address reuse to prevent port binding issues, processing of JSON-formatted client requests, and error-handling mechanisms for various network-related conditions. It also manages schedule-related operations like adding, deleting, and viewing scheduled tasks. Graceful shutdown is supported using signal handlers and atomic flags. The server is designed to operate as a persistent background service using `systemd`, ensuring that it starts automatically after installation and remains active to monitor and trigger scheduled backup tasks.

Client Implementation (SchedulerService): The client-side scheduler service provides a user-facing menu-driven interface for managing backup schedules. It is responsible for collecting user input during schedule creation, including backup type, timing information, and repository selection. It integrates with the repository service to ensure correct repository configurations and validation.

The client utilizes an instance of `SchedulerRequestManager` to communicate with the background scheduler daemon. It sends structured requests and processes responses, providing meaningful feedback and error messages to the user. This implementation ensures usability while enforcing input validation to maintain system reliability.

Network Communication Layer (SchedulerRequestManager): This component acts as the network communication bridge between the client interface and the scheduler daemon. It provides three primary methods to handle schedule creation, deletion, and viewing. The implementation uses a TCP socket connection to send JSON-formatted requests to the server.

Each request includes an `action` field that instructs the server which operation to perform, along with a `payload` field that carries necessary operation-specific data. For schedule viewing, no payload is needed; for deletion, the unique `schedule_id` must be provided; and for creation, fields such as the cron string, backup type, source path, remarks, and repository details are required. The response from the server is parsed and returned for display or further processing on the client side, ensuring effective and synchronized schedule management across the system.

5 User Interface Implementation

The application offers a dual-mode user interface comprising both a Command-Line Interface (CLI) and a Graphical User Interface (GUI). This dual design enhances usability by catering to both technical and general users, ensuring flexibility across usage scenarios.

5.1 Command-Line Interface (CLI)

The CLI functions as the primary interface for executing backup and restore operations. It is structured around a menu-driven system that guides the user through key functionalities such as repository management, backup creation, restore procedures, and integrity checks. The CLI is tightly integrated with the backend logic, providing direct access to core operations with minimal overhead. It also facilitates automation, making it well-suited for advanced or system-level usage.

5.2 Graphical User Interface (GUI)

In addition to the initially scoped CLI-based design, a fully functional GUI was also developed to enhance the system's accessibility and ease of use. This graphical interface replicates all core functionalities available in the CLI, including backup, restore, and repository management, while presenting them through an intuitive visual layout. It provides structured feedback using tables, dialogs, and progress indicators to improve user experience.

5.2.1 Technology Stack

The GUI is implemented using the Qt framework, selected for its efficiency, cross-platform capabilities, and seamless integration with C++. Key advantages include:

- Support for rapid UI development through Qt Designer.
- Availability of prebuilt widgets and layouts suitable for system-level applications.
- Smooth compatibility with the existing C++ backend.
- Minimal development overhead with a professional and responsive output.

Given the project timeline and the need for a reliable visual interface, Qt enabled fast and effective implementation without compromising functionality.

5.2.2 Integration Approach

To preserve the integrity of the original system architecture, the GUI interfaces are designed as extensions of the core CLI components. Existing logic for backup and restore was reused without modification by introducing wrapper classes tailored for GUI interaction. This approach ensures consistent behaviour across both interfaces and avoids redundancy.

The overall structure follows a separation-of-concerns model, with user interface layers (CLI or GUI) operating independently of the backend logic. This design improves maintainability and allows both interfaces to benefit from future enhancements to the core modules.

6 Technological Implementation

6.1 Technological Stack

The technical implementation of ResilioZ leverages a modern and efficient technology stack tailored for cross-platform, high-performance backup and recovery operations on Linux systems. The core application is developed in C++, chosen for its performance, system-level access, and mature ecosystem. The project utilizes the C++ Standard Library for core data structures, file system operations, and general utilities. For backup and restore operations, the system employs several specialized libraries:

- **OpenSSL** is used for cryptographic operations, including AES-256 encryption and SHA256 hashing, ensuring data security and integrity.
- **ZSTD (Zstandard)** is integrated for fast and efficient compression and decompression of backup data.
- **libnfs** is used to support NFS (Network File System) repositories, enabling seamless interaction with network-attached storage.
- **libssh** provides secure SSH and SFTP communication for remote repository support, allowing encrypted file transfers over the network.
- **libcron** is used for parsing and executing cron expressions, powering the automated scheduling system.
- **nlohmann/json** is used for JSON serialization and deserialization, facilitating structured metadata management.

The **Graphical User Interface (GUI)** is implemented using the **Qt framework**, which offers cross-platform support, a rich set of widgets, and seamless integration with C++. **Qt Designer** is used for rapid UI prototyping and layout design. Build configuration and dependency management are handled using **CMake**, ensuring portability and ease of setup across different Linux distributions. This technology stack collectively enables ResilioZ to deliver a robust, secure, and user-friendly backup and recovery solution.

6.2 Data Structures

Backup Metadata Format:

- **Backup-Level Metadata Structure**: The backup-level metadata contains essential information about the backup operation including the backup type as an integer enumeration, the precise timestamp when the backup was created, the name of the previous backup for incremental and differential backups, and optional user remarks for documentation purposes. The metadata structure follows this JSON format:

```
{
    "type": 0,                    // 0=Full, 1=Incremental, 2=Differential
    "timestamp": 1703123456,      // Unix timestamp in seconds
    "previous_backup": "",        // Name of previous backup (empty for Full)
    "remarks": "User comments",   // Optional user remarks
    "files": {}                   // File metadata dictionary
```

```
}
```

The backup type field uses integer enumeration where 0 represents a full backup, 1 represents an incremental backup, and 2 represents a differential backup. The timestamp field stores the precise creation time as a Unix timestamp in seconds, enabling accurate temporal tracking of backup operations. The `previous_backup` field contains the filename of the backup that serves as the reference point for incremental and differential operations, while the remarks field allows users to add descriptive comments about the backup purpose or context.

- **File-Level Metadata Structure**: Each file entry in the metadata contains comprehensive information about the file including the original filename, an ordered list of chunk hashes that represent the file's content, the total file size in bytes, the file modification time as a Unix timestamp, file permissions in octal format, and a SHA256 checksum of the entire file for integrity verification. The file metadata structure follows this format:

```
‘path/to/file‘ : {
    "original_filename": "document.txt",
    "chunk_hashes": [
                "a1b2c3d4e5f6...",
                "f6e5d4c3b2a1...",
                "1234567890ab..."
            ],
    "total_size": 1048576,
    "mtime": 1703123456,
    "is_symlink": false,
    "permissions": "0644",
    "sha256_checksum": "a1b2c3d4e5f6...",
    "symlink_target": "/path/to/target"     // Only for symlinks
}
```

The `original_filename` field stores the base name of the file without its full path, enabling proper restoration with the correct file naming. The `chunk_hashes` array contains an ordered list of SHA256 hashes corresponding to the chunks that make up the file, allowing accurate reconstruction during restore operations. The `total_size` field stores the uncompressed size of the file in bytes, while the `mtime` field preserves the file's modification timestamp to ensure accurate restoration of file metadata.

Additional Data Structures:

- **Maps/Dictionaries**: Used to map chunk hashes, file paths, and repository configurations for quick access.
- **Vectors/Lists/Arrays**: Store sequences like chunk hashes, file buffers, and repository record.
- **Sets**: Manage uniqueness during deduplication (e.g., unique chunk hashes).

- **Custom Structs and Classes**: Used for encapsulating metadata, backup info, repository config, and scheduling. Examples include `FileMetadata`, `BackupDetails`, and `ScheduleEntry`.
- **Queues**: Manage pending backup/restore tasks in the scheduler.
- **Buffers**: Used for reading/writing chunk data during compression and transfer.
- **JSON Objects**: Represent configuration and metadata using `nlohmann::json`.

6.3 Algorithms

6.3.1 FastCDC

The FastCDC (Fast Content-Defined Chunking) algorithm represents an approach to content-defined chunking that provides excellent deduplication efficiency while maintaining high performance characteristics. This algorithm uses a sophisticated sliding window approach with gear hashing to identify optimal chunk boundaries based on content characteristics rather than fixed positions. The algorithm implements a three-region approach with different hash masks and processing strategies for each region. The minimum region ensures chunks meet minimum size requirements, the normal region uses a smaller mask for more frequent boundary detection, and the maximum region uses a larger mask to prevent excessively large chunks.

Gear Hashing Mechanism: The gear hashing function provides the mathematical foundation for boundary detection in the FastCDC algorithm. This function uses a precomputed table of 256 random 32-bit values that serve as lookup values for each possible byte value, eliminating the need for complex mathematical operations during hash computation. The gear hash function processes data byte by byte, using each byte as an index into the `GEAR_TABLE` to retrieve a random value. These values are combined using bit shifting operations to produce a final hash value that exhibits excellent distribution properties for boundary detection. The algorithm maintains a 64-byte sliding window for hash calculation, updating the hash incrementally as the algorithm progresses through the data. This approach provides constant-time hash updates while maintaining the statistical properties required for effective boundary detection.

Boundary Detection Process: The boundary detection process implements sophisticated logic across three distinct regions with different characteristics and processing strategies. The minimum region ensures chunks meet minimum size requirements by delaying boundary detection until the minimum size threshold is reached. The normal region uses a 13-bit mask for boundary detection, providing frequent boundary identification that optimizes deduplication efficiency. The algorithm processes two bytes at a time in this region when possible, reducing the number of iterations and improving performance. The maximum region uses an 11-bit mask to prevent excessively large chunks that could impact storage and processing efficiency. This region processes data byte by byte to ensure precise boundary detection while maintaining performance characteristics.

Performance Optimizations: The FastCDC implementation includes several performance optimizations that enhance processing speed while maintaining deduplication efficiency. The dual-byte processing in the normal region reduces iteration count and improves throughput for large files. The sliding window approach

provides constant-time hash updates regardless of window size, ensuring consistent performance across different file sizes and content types. The incremental hash updates eliminate the need to recalculate the entire hash for each position, significantly improving processing speed. The algorithm uses different hash masks for different regions to optimize boundary detection characteristics for each region's requirements. This adaptive approach ensures optimal deduplication efficiency while maintaining reasonable chunk sizes and processing performance.

6.3.2 SHA256 Hash

Hash Function Overview: The SHA256 (Secure Hash Algorithm 256-bit) implementation enhances both data integrity and deduplication across the backup system. It generates a unique 256-bit (32-byte) cryptographic hash for each file, which is stored in the backup metadata and used during restoration to verify that the restored file matches the original. Additionally, chunk and compressed chunk hashes are embedded in their filenames to support content-addressable storage and efficient chunk identification, enabling deduplication and consistency checks during reconstruction, even though individual chunk integrity is not separately verified.

File-Level Hashing: The system calculates SHA256 hashes for individual files to provide integrity verification and change detection capabilities. The `CalculateFileSHA256` function implements efficient hash calculation using OpenSSL libraries with buffered reading to handle large files without excessive memory usage. The function processes files in 4KB chunks to maintain reasonable memory usage while providing efficient processing for files of any size. The implementation uses the OpenSSL `SHA256_CTX` structure to maintain hash state across multiple read operations, ensuring accurate hash calculation for large files. The file-level hashes are stored in file metadata and used for change detection during incremental and differential backups. These hashes enable the system to quickly identify changed files without performing content-based comparisons, significantly improving backup performance.

Chunk-Level Hashing: Each chunk receives a SHA256 hash based on its content, providing unique identification for deduplication. The chunk hashes are calculated after compression to ensure that the hash reflects the actual stored content. The chunk hashing process uses the same OpenSSL implementation as file hashing, ensuring consistency and reliability across all hash calculations. The hashes are stored in file metadata as a vector of strings, enabling the system to reconstruct files from their constituent chunks during restore operations. The chunk-level hashes enable deduplication by identifying identical chunks across multiple files and backups. This capability provides significant storage savings, particularly for incremental backups where many chunks remain unchanged between operations.

Hash-Based Storage Organization: The system uses chunk hashes for intelligent storage organization to prevent filesystem performance degradation when storing large numbers of small files. Chunks are organized in subdirectories based on the first two characters of their hash, distributing storage load across multiple directories. This hash-based organization provides several benefits including improved filesystem performance, better load distribution, and simplified chunk management. The system automatically creates subdirectories as needed and maintains consistent organization across all storage operations.

6.3.3 AES-256 Encryption

AES-256 (Advanced Encryption Standard with a 256-bit key) is a symmetric block cipher that encrypts and decrypts data using the same secret key. As part of the AES family standardized by NIST, AES-256 offers the highest level of security among its variants (AES-128, AES-192, AES-256) due to its longer key length. It operates on 128-bit blocks of data and performs 14 rounds of transformation steps including substitution, permutation, mixing, and key addition. AES-256 is widely used for securing sensitive data due to its strong resistance to brute-force attacks and its efficient implementation on both software and hardware platforms.

Key Derivation and Initialization: AES-256 requires a secure 256-bit (32-byte) key and a 128-bit (16-byte) Initialization Vector (IV) for CBC (Cipher Block Chaining) mode. Since user-supplied passwords are typically shorter and less secure than cryptographic keys, key derivation is performed using PBKDF2 (Password-Based Key Derivation Function 2). This function uses HMAC-SHA256 and a random salt to derive a strong, unique key from the password. A random IV is also generated to ensure that identical plaintexts encrypted with the same key produce different ciphertexts, preventing pattern analysis.

Encryption Process: The encryption process follows these key steps:

1. **Padding:** The plaintext is padded using a standard scheme (e.g., PKCS#7) to ensure its length is a multiple of the AES block size (16 bytes).
2. **Key and IV Preparation:** The key is derived using PBKDF2 from the password and salt, while the IV is randomly generated.
3. **Context Initialization:** An AES cipher context is created and initialized with the AES-256-CBC mode, key, and IV.
4. **Encryption:** The padded plaintext is encrypted in blocks, producing ciphertext.
5. **Finalization:** The final block is processed and appended, completing the encryption.

The output consists of metadata (magic bytes, salt, IV) followed by the ciphertext, allowing the decryption process to reconstruct the original state.

Decryption Process: Decryption follows the inverse process:

1. **Header Parsing:** The algorithm extracts the magic bytes, salt, and IV from the input buffer.
2. **Key Regeneration:** Using the same password, salt, and PBKDF2 parameters, the original key is re-derived.
3. **Context Initialization:** A cipher context is created and initialized with AES-256-CBC using the re-generated key and IV.
4. **Decryption:** The ciphertext is decrypted to yield the padded plaintext.
5. **Unpadding:** The original plaintext is recovered by removing padding bytes.

If the password is incorrect or data is corrupted, the decryption will fail during finalization due to authentication mismatches.

Security Features: AES-256 provides robust security guarantees:

- **Strong Key Length:** 2^{256} possible keys, making brute-force infeasible.
- **Random Salt & IV:** Prevents precomputation attacks and ensures ciphertext uniqueness.
- **PBKDF2 Derivation:** Defends against brute-force attacks on weak passwords by introducing computational cost.
- **Magic Bytes:** Embedded in encrypted data to distinguish encrypted metadata from plaintext or corrupted content.

6.3.4 Zstandard (ZSTD)

Zstandard (ZSTD) is a modern lossless compression algorithm developed by Facebook, offering a great balance between compression ratio and speed. Designed for real-time applications, ZSTD features tunable compression levels, fast decompression speeds, and support for dictionary-based compression. It is well-suited for scenarios such as backup systems, where both speed and space savings are critical. The algorithm uses entropy coding techniques such as Finite State Entropy (FSE) and Huffman coding to achieve high compression ratios while remaining CPU-efficient.

Compression Workflow: The compression process in this implementation wraps the ZSTD compressor with metadata encoding for chunk restoration:

1. **Determine Maximum Output Size:** The maximum size of the compressed output is calculated using `ZSTD_compressBound()`. This ensures the destination buffer is large enough to hold any possible compressed output for the given input size.
2. **Allocate Output Buffer:** A `std::vector<uint8_t>` buffer is created large enough to store both the maximum compressed data and an additional prefix to store the original (uncompressed) size.
3. **Store Original Size:** The original size of the chunk is stored in the first `sizeof(size_t)` bytes of the buffer. This is critical for decompression, which needs to know the exact output size in advance.
4. **Perform Compression:** Compression is performed using `ZSTD_compress()`, which takes the input data, maximum destination buffer size, and default compression level. The compressed data is written after the size prefix.
5. **Resize Output:** After compression, the buffer is resized to the actual size: `sizeof(size_t)` for the prefix plus the actual compressed size returned by the compressor.
6. **Assign SHA256 hash as name:** A SHA256 hash of the compressed data is taken. This helps uniquely identify the chunk by the metadata of the system for restoration and verification purposes

This approach ensures each compressed chunk is self-contained and easily decompressible without requiring separate metadata files.

Decompression Workflow: The decompression logic mirrors the compression pipeline and restores the original data using the embedded size header:

1. **Read Original Size:** The first `sizeof(size_t)` bytes of the input buffer are interpreted as the original (decompressed) size.
2. **Prepare Output Buffer:** A buffer of the exact decompressed size is allocated using `std::vector<uint8_t>`.
3. **Decompress:** The actual compressed payload (data following the prefix) is passed to `ZSTD_decompress()`, which decompresses it into the pre-allocated buffer.
4. **Error Handling:** If decompression fails, an error message is generated using `ZSTD_getErrorName()` and propagated using a utility function.

The decompression process is deterministic, relying solely on the compressed data and the embedded size prefix—no external metadata or state is required.

Performance Characteristics: Zstandard is optimized for real-time compression and decompression:

- **High Throughput:** ZSTD offers decompression speeds upwards of 500 MB/s on modern CPUs, ideal for streaming backup systems.
- **Configurable Compression Levels:** Users can trade compression ratio for speed by adjusting compression levels (e.g., `ZSTD_CLEVEL_DEFAULT`).
- **Bounded Memory Usage:** ZSTD provides guarantees about the maximum buffer size needed via `ZSTD_compressBound()`, allowing predictable memory allocation.
- **Minimal Latency:** Fast decompression with low CPU overhead makes ZSTD suitable for restore-time performance-sensitive applications.

7 Features and Functionality

ResilioZ offers a comprehensive suite of features designed to provide robust, flexible, and user-friendly backup and recovery capabilities for Linux environments:

7.1 Backups of Types (Full, Incremental, Differential)

The system supports full backups, which capture complete snapshots of all selected data, as well as incremental and differential backups that optimize storage and performance by only processing files that have changed since the last backup or the last full backup, respectively. This flexibility allows users to tailor their backup strategies to their operational needs and storage constraints.

7.2 Restore Functionality

The restore engine enables users to recover data from any backup type, reconstructing files and directories from compressed, chunked storage. The system supports streaming restoration for large files, preserves original directory structures, and restores file metadata such as permissions, timestamps, and symbolic links, ensuring accurate and reliable data recovery.

7.3 Integrity Verification

Data integrity is maintained through the use of SHA256 checksums at the file levels. During backup verification and restore operations, the system verifies these checksums to detect any corruption or tampering, providing users with confidence in the reliability of their backups.

7.4 Repository Management

The Repository Service abstracts the management of local, NFS, and remote (SSH/SFTP) repositories, allowing users to create, select, and delete repositories securely. It handles password verification, maintains a central repository metadata registry, and ensures consistent repository management across all storage backends.

7.5 Scheduling Backups

Automated backup scheduling is provided through a client-server architecture using the `libcron` library. Users can create, view, and delete scheduled backup jobs using cron expressions, ensuring that backups occur regularly and without manual intervention. The scheduler logs all activities for audit and compliance purposes.

7.6 GUI Interaction

In addition to the CLI, ResilioZ features a fully functional GUI built with Qt. The GUI provides intuitive visual workflows for all core operations, including backup, restore, repository management, and scheduling. It uses dialogs, tables, and progress indicators to enhance usability and accessibility for a wide range of users.

7.7 Metadata Handling

The system maintains detailed, encrypted metadata for every backup, including backup type, timestamps, file lists, chunk references, and file attributes. Metadata is stored in JSON format and protected with AES-256 encryption, ensuring both transparency for restore operations and security against unauthorized access.

8 Setup

8.1 Build Setup

1. Install Dependencies

The following dependencies need to be installed to build the application:

Build Tools

- `g++` – C++ compiler
- `cmake` – Build system generator
- `make` - To build executables
- `git` – Version control & installing libcron
- `pkg-config` – Helper tool for managing compiler and linker flags

Development Libraries

- `libssh-dev`: SSH protocol support
- `libzstd-dev`: Zstandard compression support
- `libnfs-dev`: NFS client library
- `qt6-base-dev`: Qt 6 base development files for GUI
- `libxkbcommon-dev`: Keyboard handling support for Qt
- `zlib1g-dev`: zlib compression support (libssh dependency)
- `libcron`: C++ cron-style job scheduler (Installed using git by CMake)

2. Clone the Repository

Clone the repository to your local development environment from the GitHub Repository.

3. Run `run.sh`

Run the `run.sh` file in the repo, The required executables will be found in the `/build` folder.

8.2 External Repository Setup

8.2.1 NFS Repository

NFS Server Setup:

1. Install the NFS Server: `sudo apt install nfs-kernel-server`
2. Edit the exports file to define shared directories: `sudo nano /etc/exports`

3. Add a line for the directory you want to share: `/path/to/backup [IP Address (or) hostname (or) *(for all IPs)](rw,sync,no_subtree_check,no_root_squash))`
4. Apply the new export rules: `sudo exportfs -ra`
5. Set permissions on the shared directory: `sudo chmod -R 777 /path/to/backup`

NFS Client Setup:

1. Install client utilities: `sudo apt install nfs-common`

8.2.2 Remote Repository

SSH Server Setup

1. Install SSH Server: `sudo apt install openssh-server`
2. Enable the SSH Server: `sudo systemctl enable ssh`
3. Make sure to edit in: `/etc/ssh/sshd_config`

   ```
   PubkeyAuthentication yes
   AuthorizedKeysFile .ssh/authorized_keys
   ```

4. Restart the SSH Server: *sudo systemctl restart ssh*

SSH Client Setup

1. Install SSH Client: `sudo apt install openssh-client`
2. Make sure the `/home/[username]/.ssh` folder has required read/write permissions
3. Generate an SSH key on your client system and save it in `/home/[username]/.ssh/[identifier]`
   E.g. `ssh-keygen -t rsa -b 4096 -C "identifier"`
4. Copy the public key in `[keyname].pub` to the server's `authorized_keys`
   E.g `ssh-copy-id username@[server IP (or) hostname]`

9 Conclusion

The ResilioZ Linux Backup and Recovery System successfully delivers a comprehensive, modular solution for data protection. Through its support for full, incremental, and differential backups, robust restore functionality, and integrity verification using SHA256, the system ensures reliable and efficient safeguarding of user data. The architecture's repository abstraction allows seamless management of local, NFS, and remote (SSH/SFTP) storage backends, while the integrated scheduling system automates regular backups, reducing the risk of human error and missed backup windows. Both CLI and GUI interfaces provide accessible, user-friendly ways to manage all core operations, and the use of strong encryption and detailed metadata handling ensures that data remains secure and recoverable across a variety of real-world scenarios. During development, challenges included ensuring consistent performance across different storage environments, handling large files efficiently, and maintaining data integrity during backup verification and restore operations. Integrating multiple third-party libraries (for compression, encryption, network storage, and scheduling) required careful attention to compatibility and error handling. Looking ahead, there is scope to further enhance the system by introducing features such as network cache resume, which would allow interrupted transfers to continue from the point of failure, improving reliability for remote and network-based backups. Additional future work could include expanding automated testing, refining the user interface, and supporting more advanced backup policies to meet evolving enterprise needs.

10 References

1. OpenSSL Official Website – the primary site for the OpenSSL project, offering downloads, documentation, and library information.
2. facebook/zstd on GitHub – the main repository for Zstandard, Facebook's fast real-time compression algorithm.
3. sahlberg/libnfs on GitHub – an NFS client library in C for Linux and other UNIX-like systems.
4. libssh Official Website – the home page for libssh, a multi-platform C library implementing SSHv2 client and server functionality.
5. libcron on GitHub – a C++ scheduling library using cron-style expressions.
6. nlohmann/json on GitHub – JSON for Modern C++, a popular single-header C++ JSON library.
7. Qt Official Website – the official site for the Qt framework, offering tools and product downloads for GUI application development.
8. Xia W, Zhou Y, Jiang H, Feng D, Hua Y, Hu Y, Liu Q, Zhang Y. **FastCDC**: A fast and efficient Content-Defined chunking approach for data deduplication. In 2016 USENIX Annual Technical Conference (USENIX ATC 16) 2016 (pp. 101-114).