# 18746 Project - CloudFS Implementation Report

Siqi (Edward) Guo, AndrewID: siqiguo

December 10, 2024

## Introduction

As the demand for scalable, efficient, and cost-effective storage solutions grows, the integration of cloud storage with local file systems has emerged as a promising approach. CloudFS, a cloud-backed local file system, bridges the gap between high-performance local storage and the virtually infinite capacity of cloud services. By taking advantages of local solid-state disks (SSD) and cloud storage, CloudFS aims to deliver a seamless, efficient, and robust storage solution.

In this project, we developed single-threaded CloudFS, using the FUSE framework, and incorporating S3 service API, `archive-lib`, `libfuse`, and several other useful libraries.

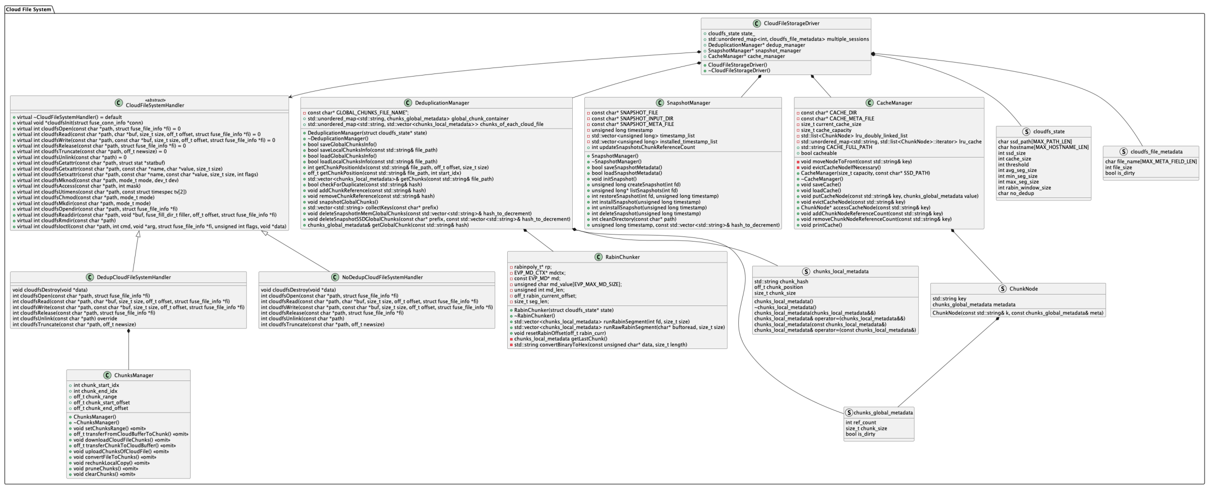
## System Overview

CloudFS, and by extension this project, consists of four components: a core file system leveraging the properties of local SSD and cloud storage for making data placement decisions; a component that takes advantage of redundancy in data to reduce storage capacity; a component that adds the ability to create, delete, and restore file system snapshots while minimizing cloud storage costs; and a component that uses local on-device and in-memory caching to improve performance and reduce cloud operation costs.
Based on the above hints from the handout, I designed the whole CloudFS with four following main components.

1. **Cloud File System Handler**: This module serves as the interface for file operations, managing metadata and coordinating data placement across local and cloud storage.
2. **Deduplication Manager**: This component holds the chunk's state. By identifying and eliminating duplicate data, it minimizes storage usage and optimizes data transfer. It also maintains a Chunk Manager to manage the file system buffer and assist itself to upload and download chunks.
3. **Cache Manager**: This module implements caching strategies to store frequently accessed data locally, reducing cloud access latency and costs.
4. **Snapshot Manager**: This component provides versioning capabilities, enabling users to create and manage snapshots efficiently.

### Cloud File System UML Diagram

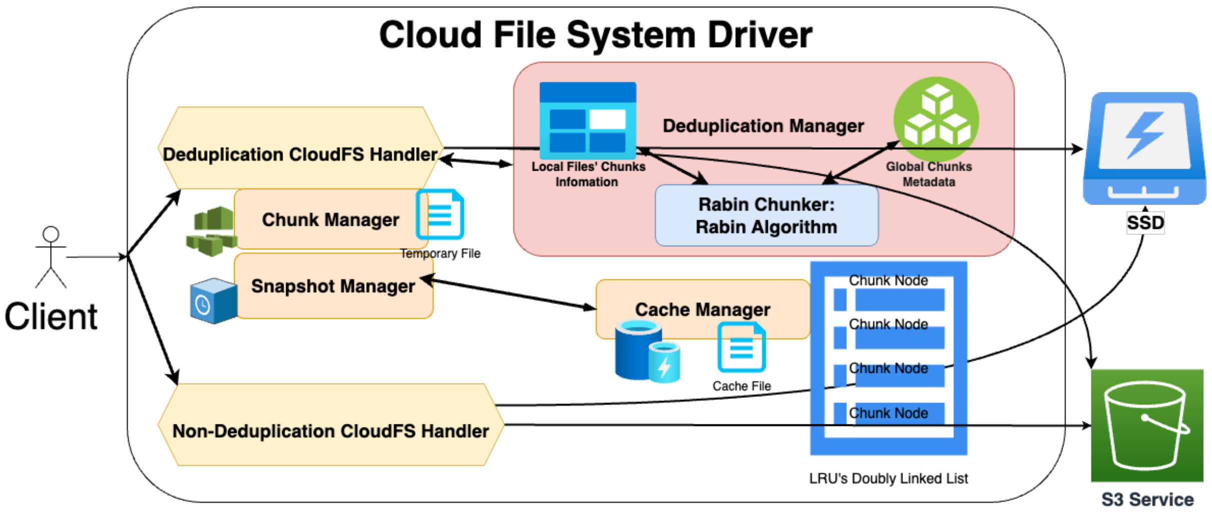
Here is the UML diagram of the CloudFS design. The following diagram illustrates the architecture of CloudFS, highlighting the relationships between its main components and their interactions.



### CLouFS Components Design and System Framewrok

The second figure complements this by elaborating on the responsibilities of each component, offering a detailed perspective on how they contribute to the system's operations. Together, these figures provide a comprehensive view of the CloudFS architecture.

Here is the role of each component in the CloudFS:



## Detailed Design, Policies, and Justifications

### Checkpoint 1: Hybrid file system spanning SSD and cloud storage

In the first checkpoint of this project, I implemented the framework of the CloudFS, a simple hybrid file system with two different storage components: a local SSD and a cloud storage service similar to Amazon S3.

**Data Placement.** The necessary metadata of a file will be stored on the local SSD and the file content will first be placed on local device. The file will be transferred to the cloud storage when this file grows beyond a threshold and the file system namespace on the SSD will be updated using a user-defined link. Similarly, the file content will be downloaded from the cloud storage to the local device when that file shrinks below the threshold and the cloud storage will be cleared to save more space.

**Handler Implementation.** CloudFS is a FUSE-based file system, most of the code will be written to implement functions called through the VFS interface. I built a working prototype using a subset of VFS calls including `getattr`, `getxattr`, `setxattr`, `mkdir`, `mknod`, `open`, `read`, `write`, `release`, `opendir`, `readdir`, `init`, `destroy`, `access`, `utimens`, `chmod`, `link`, `symlink`, `readlink`, `unlink`, `rmdir`, and `truncate`.

**Metadata.** One necessary file metadata will be the file's size, which determines its storage location. CloudFS-specific metadata is stored in the file's extended attributes and is consistently updated on non-volatile storage to ensure reliability. Additionally, a mapping is maintained between the local file identifier and its corresponding storage position in the cloud. To minimize overhead, I implemented a fixed mapping scheme, eliminating the need for persisting this mapping in checkpoint 1.

### Checkpoint 2: Block Level Deduplication

Recall that the cloud storage, request to the Cloud, and data transfer via Cloud Service are expensive, so we need to reduce the storage usage and communication costs via Cloud API as much as possible. We notice that the data redundancy is a common issue in the file system, so we decide to implement a deduplication manager to eliminate the data redundancy, reduce the storage usage, and optimize data transfer.

**Granularity of Deduplication.** In this checkpoint, I implemented a block-level deduplication scheme. The file content is divided into similar-size segments (blocks), and the deduplication manager identifies these duplicate blocks and stores them only once. The deduplication manager maintains a hash table to track the unique blocks and their global reference counts. This lookup table will be used to search for duplicate segments. Another table is used to track the identifiers (e.g. MD5 of the chunk) of all segments and all necessary metadata of those corresponding segments in each local file.

**When and How to Chunk the File.** The entire file will be chunked into blocks when it is larger than the threshold for the first time. If a file is known to be on the cloud, the file's relevant segments will be downloaded first no matter for the consequent `read` or `write`. After a `write`, the local copy of those downloaded content will be modified and rechunked into new blocks. The chunking process is done by the Chunk Manager with Rabin's algorithm. As Rabin calculates the hash value of the segment based on the content, the same contents in any files will be actually stored once. Multiple identical segments will only increment the global chunk reference count. In this way, the files in the CloudFS get deduplicated to the largest extent.

**Trade-off between cloud usage and data transfer costs.** Ideally, after write, Chunker needs to keep downloading consequent segments to construct continuous blocks in the buffer, and rechunking until the Chunker finds an existing boundary. However, I simplified the process by only chunking the downloaded segments, which is not the best choice, but is efficient enough for the current implementation. The trade-off is between the efficiency of deduplication and the cost of data transfer. Without precise deduplication, the cloud usage will be increased, but the data transfer costs will be minimized. As the data transfer is more expensive, I intended to reduce the data transfer costs with my simplification.

**The Right Average Segment Size for Deduplication.** Larger the segment size, less cost to maintain the lookup table, faster to search the deduplicated chunks, but less likely to find duplicates. The average segment size should be chosen based on the trade-off between the deduplication efficiency and the overhead of maintaining and searching the hash table. In my implementation, I chose 4KB as the average segment size, which is a common choice in the industry and this project.

**Checkpoint 3: Snapshots & Cache in CloudFS**

**Snapshots Management.** In this checkpoint, I implemented the Snapshot Manager to support the IOCTL functions in CloudFS. The Snapshot Manager maintains a list of snapshots for each file and provides functionalities to create, delete, restore, install, and uninstall snapshots.

Creates a snapshot of the CloudFS by archiving the snapshot input directory and uploading it to the cloud. Updates deduplication information to pin chunks associated with the snapshot.

Restores a snapshot by retrieving the associated tar file from the cloud and extracting it. Clean the old directory before extracting the tar file to restore one snapshot.

Deletes a snapshot by deleting the associated tar file from the cloud and updating all snapshots' metadata.

Other functions like `install`, `list`, and `uninstall` are also implemented to support the snapshot management.

**Cache Management.** To further optimize the CloudFS performance and reduce the cloud costs, I implemented the Least Recently Used (LRU) write-back cache in the CloudFS. Cache will store the frequently accessed chunks locally to reduce the cloud access frequency and costs. In this case, uploading will only happen when there is an eviction and downloading will only happen when the chunk is not found in the cache.

What is worth mentioning is that the cache policy is implemented very easy in this checkpoint, as my unintentional design in the previous checkpoints is well extensible to support the cache. Previously, the Deduplication Manager has already relied on the Chunk Manager to manage the buffer for uploading and downloading chunks one by one, now I just need to redirect the buffer to the cache file to maintain the separate chunk as the cache node in the cache container.

Given the cache policy must have no memory-based component, the cache metadata also needs persistency across mounts, just as the same as snapshot metadata and chunk metadata.

**Performance Improvement.** After implementing the cache, the CloudFS performance is significantly improved. We could observe a reduced cost from 28.7$ to 15.4$ in the provided test cases on the AutoLab.

## Conclusion

In this project, I implemented a CloudFS that bridges the gap between local SSD and cloud storage, featuring a hybrid-scheme file placement, block-level deduplication, snapshots, and cache management.