Contents

Overview

For this project your mission will be to build CloudFS, a cloud-backed local file system. Specifically, a CloudFS instance stores all file metadata and some file data on a local Solid State Disk (SSD), and the rest of the file data using a cloud storage service such as Amazon S3, Microsoft Azure, Dropbox, etc. To make the development process easier, you will be building CloudFS using the FUSE (Filesystem in USErspace) software interface.

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. You have already built the first two components in the previous two checkpoints. In Checkpoint 3 you will be focusing on the latter two. The deadline to submit your CloudFS project report is **11:59pm EST on December 11, 2020**. To help ensure you progress steadily through the project goals, we have imposed three **graded** intermediate milestones involving source code submissions:

Milestone	**Description**	**Deadline**
Checkpoint 1	Hybrid FUSE file system	11:59pm EST on Monday November 4, 2020
Checkpoint 2	Deduplication	11:59pm EST on Monday November 17, 2020
Checkpoint 3	Snapshots and Caching	11:59pm EST on Friday December 11, 2020
Report	Final Project Report	11:59pm EST on Friday December 13, 2020

The last submission you make for Checkpoint 3 will be your final code submission. Note that there are a couple of days between the final code hand-in and the final project report hand-in. This will leave you a little time to think through and carefully write your project description and performance evaluation report. You must submit your code to Autolab for each milestone; some automated testing will be available through Autolab, but all sections will be evaluated by more than the automated tests.

Also note that you **cannot** submit the Final report late – you must submit the final report on or before December 13, 2020.

Note: you are allowed 25 submissions per checkpoint. Any additional submission after the first 25 submissions will incur a penalty of 10% of the checkpoint grade. Autolab shows infinite submissions because we do not impose a hard limit, but rather have a penalty per submission. Furthermore, you have a total of three grace days (i.e., unpenalized late days) for the entire semester. Each checkpoint has a due date and an end date. Grace days can be used to avoid late penalties for submissions past the due date. There will be a late penalty of 10% of the checkpoint grade per day if you run out of grace days. **No submissions are accepted after the end date**.

The rest of this document describes the specification of CloudFS in the context of the third, and last, project checkpoint. In Section 1 we describe the first part of Checkpoint 3. For that, you will be building on your CloudFS implementation from Checkpoint 2. Your goal will be to add functionality for managing consistent file system snapshots. Section 2 will then describe the second part of Checkpoint 3, which requires you to utilize a local cache to reduce the cloud costs incurred by your CloudFS implementation. As in the

previous handout, Section 3 provides information on the starter code you received, and Section 4 describes the expected project deliverables.

1 Checkpoint 3, Part One – Snapshots

So far, you have built a file system for hybrid cloud storage. Furthermore, your file system (hopefully) utilizes cloud storage efficiently by fragmenting files into segments that make it easy to detect duplicate content across (and within) files. For the first part of this checkpoint you will be extending the functionality of your CloudFS implementation to include the ability to restore the file system to a previous state, i.e., an earlier point in time. This feature can be used to generate consistent CloudFS backups which, as we have seen in class, come in handy when we need to undo a mistakenly deleted or corrupted file, and when we need to restore the file system on a new SSD after a device failure. We assume that data residing on the cloud is reliably stored, but that is because we are paying for that reliability guarantee (so it becomes someone else's problem). These consistent CloudFS backups are snapshots of the file system's state at a given point in time, and you have to extend your implementation to create, and restore from such snapshots. Because we have assumed that cloud storage is reliable, you will be storing your snapshots in the cloud.

The process of taking a snapshot involves capturing the state of the entire file system at a given point in time and copying said state to the cloud. After the snapshot is taken, the file system should be available again for normal use. When a snapshot is restored, all changes since the snapshot was taken should be undone, returning the file system to that point in time.

In addition to creating and restoring snapshots, you will need to implement functionality for *installing* them. By installing a snapshot, the user must be able to browse the content of an existing snapshot without restoring it and altering the state of the file system. This is reminiscent of the `mount` operation in Linux, although it does not involve the creation of a new VFS entry. Note that the snapshots are read-only, i.e., immutable objects, so no modifications are permitted to installed snapshots.

1.1 Implementing Snapshots in CloudFS

The simplest way to implement snapshotting programmatically would be to copy every bit from the local SSD to the cloud. Then, when a restore operation is performed, every bit is copied back to overwrite the contents of the local SSD. As you can imagine, this would be a grossly inefficient implementation: it requires every bit to be accessed, even those on the cloud that are assumed to be reliably stored already. As a result, it is expected to incur high cloud costs and take a significant amount of time to complete proportional to the file system capacity. Thus, we will consider a better approach that leverages the functionality you have already implemented.

Since you completed Checkpoint 2, your CloudFS implementation already stores some data in the cloud's reliable storage, some on the local disk, and some metadata that stitch the two storage backends in one consistent namespace. More specifically, one of the data structures you use stores reference counts that determine the number of times each cloud-backed data segment is referenced across the entire file system. Using this information, we will consider an alternative approach to implementing snapshots. Our approach will have to handle the following information:

- **File data.** The data for all files larger than the migration threshold already resides in the cloud, so we

do not need to make a copy of that data. We do, however, need to handle the data of all the small files that reside fully on the local disk.

- **Metadata.** All the metadata that resides on the local disk, which map local and cloud data into a consistent namespace, must be copied during the snapshot creation process.
- **Reference Counts.** The reference counts for every data segment stored in the cloud will obviously need to be copied as part of the file system metadata. After doing so, however, these counts will have to be manipulated to ensure that all data segments referenced by the snapshot are not deleted from the cloud storage while the snapshot is still available.

One way to ensure that data segments will not be removed prior to the snapshot itself, could be by incrementing their reference counts by one. This, coupled with a copy of the file system's metadata and the data of small files stored on the local disk would constitute a valid snapshot, provided that the copy and reference count updates are performed as an atomic operation. This should be easy to achieve in your single-threaded FUSE implementation.

1.2 Glossary and Tools

This section defines terms related to tools that you will be using as part of your implementation of snapshotting in CloudFS.

- **`ioctl` call.** `ioctl` is an abbreviation of "Input/Output ConTroL". It can be viewed as a wildcard system call that can be used to implement operations that are not standard enough to warrant the creation of a dedicated system call. They are also a vehicle for extensibility as they can be used to add operation support without making it necessary to extend the system call interface. This is how we will be using them in the context of snapshots.

  An `ioctl` takes three arguments: a special file descriptor (which we will cover next), an integer `cmd` that represents the code of the requested command, and a `void *` pointer to data passed with the command. When defining commands supported by the `ioctl` call you will also be specifying whether data is also expected, and the type of said data. Then, the underlying layers will handle the process of copying this data into the kernel and back out to userspace. Command definition is achieved through macros, and they can be found in `src/snapshot/snapshot_api.h`.

  The prototype of the `ioctl` call that will be received by CloudFS, and therefore must be implemented by you, is as follows:

  ```
  int cloudfs_ioctl(const char *path, int cmd, void *arg,
                    struct fuse_file_info *fi, unsigned int flags,
                    void *data)
  ```

  Of the above, you have to handle the `cmd` and `data` arguments. Once you have done that, you can use the command line program we provide (see `src/snapshot/snapshot-test.c`) to test your implementation.

- **Snapshot timestamp.** We define a given snapshot's timestamp as the value (in microseconds) that is returned by a `gettimeofday()` call during the creation of the snapshot. A microsecond granularity

should be fine enough to act as a unique identifier across snapshots[1].

- **`.snapshot` file.** A special file that is always present in the root directory of your FUSE mount. The file descriptor resulting from an `open` call on this file can be used with the CloudFS `ioctl` call to open a communication channel with CloudFS.

1.3 Interface Specification

All snapshots are identified by their timestamp, a 64-bit unsigned integer value. The file system should have the capability to handle a predefined number of snapshots, namely `CLOUDFS_MAX_NUM_SNAPSHOTS` (the definition can be found in `src/snapshot/snapshot-api.h`.

Your snapshot implementation should support the following operations:

- **`SNAPSHOT`**: Copies the minimum set of data required to take a snapshot and store it in the cloud. `SNAPSHOT` can be called multiple times. Invoking `SNAPSHOT` after the maximum number of allowed snapshots has been reached (i.e., `CLOUDFS_MAX_NUM_SNAPSHOTS`) should return an error. In case there are currently installed snapshots when a `SNAPSHOT` operation is initiated, the file system should return an `ioctl` error.
- **`RESTORE`**: Allows you to restore from a specified snapshot. All file system state (i.e., metadata and data) changes since the snapshot was captured should be undone by the restore operation. In addition, all the snapshots that were taken following the restored one should also be discarded.
- **`DELETE(t)`**: Deletes the snapshot identified by timestamp *t* and frees up any resources occupied by that snapshot without affecting other snapshots or the state of the live filesystem. Deleting an installed snapshot should return an `ioctl` error.
- **`INSTALL(t)`**: Should "install" the snapshot identified by timestampt *t* in a read-only folder that resides in the FUSE root directory under the name `snapshot_t`, where `t` should be replaced by the timestamp value. Multiple invocations of an already installed snapshot, or other invalid install requests should return appropriate `ioctl` errors. However, multiple snapshots can be installed in their respective `snapshot_t` folders simultaneously.
- **`UNINSTALL(t)`**: Should "uninstall" the snapshot identified by the given timestamp *t*. In case the requested snapshot is not already installed, the file system needs to return an error via the `ioctl` call.
- **`LIST`**: Returns a list of timestamps (i.e., 64-bit unsigned integer values) which are all the snapshots currently present in the cloud.

These operations are executed via `ioctl` calls on a special file that should appear in the root directory of CloudFS. This file is named `.snapshot` and it has the following properties:

- It can only be opened in read-only mode
- It is always present in the file system
- Delete, read, and write operations on the file fail to execute

[1] This means we don't expect you will be able to take multiple snapshots within a microsecond, but you're welcome to prove us wrong!

- The file size is zero
- It should be the only file in the FUSE root directory that allows `ioctl` calls

The `.snapshot` file provides the file descriptor (via the `open` call) that can be passed to `ioctl` calls. This allows applications in userspace to access the snapshot API that we defined above. You should make sure that you modify the right set of VFS operations in order to allow the `.snapshot` file to have these features.

As mentioned above, `ioctl` calls are a general tool that allows FUSE, device drivers, file systems, etc. to be extended with custom functionality without having to insert additional syscalls or callbacks to the existing interface. You are strongly encouraged to read the FUSE `ioctl` specification carefully, and are required to strictly adhere to the `ioctl` definitions currently present in `snapshot-api.h`

1.4 Implementation Roadmap

The goal here is to implement a FUSE `ioctl` function that executes all six snapshot operations described above, depending on the value of the `cmd` argument passed to the `ioctl` call.

The following *10 easy steps!*™ describe a potential roadmap you could follow to implement snapshotting functionality in CloudFS:

1. Read and understand `ioctl` calls, FUSE `ioctl` calls, and the contents of `snapshot-api.h`
2. Implement changes to the existing CloudFS FUSE operations to expose the `.snapshot` file according to the provided specification. Test `.snapshot` by calling `open` in `RDONLY` mode.
3. Implement a dummy `ioctl` operation and test it to ensure that `ioctl` calls are reaching the right location. This operation does not have to be one of the operations listed above, as it is only meant for testing your code.
4. Implement the `SNAPSHOT ioctl` operation by copying metadata and small files to the cloud, and then updating the reference counts of the data segments stored already in the cloud.
5. Generate a snapshot and test it by restoring it manually.
6. Implement the `RESTORE ioctl` operation and manually verify its correctness.
7. Implement the `DELETE ioctl` operation and test that it can actually delete existing snapshots.
8. Implement the `LIST ioctl` operation and test its ability to list all existing snapshots.
9. Implement the `INSTALL` and `UNINSTALL ioctl` operations and test that they provide and remove access to snapshot content as intended.
10. Start running the test scripts and then proceed to more rigorous testing by writing your own test cases.

1.5 Testing and Evaluation

Your implementation of Checkpoint 3 will be tested for both correctness, and cache performance.

Correctness is defined as the ability of your implementation to materialize a snapshot interface with the semantics we described in the provided specification.

Cache Performance is defined as the ability of your implementation to avoid incurring unnecessary cloud capacity, operation, and transfer costs.

2 Checkpoint 3, Part Two – Caching

Our goal in this part of Checkpoint 3 will be to further improve the efficiency of your CloudFS implementation! During Checkpoint 2, we attempted to leverage duplication in file system data to reduce the costs incurred by the cloud storage component of CloudFS. This time you will attempt to further reduce cloud costs by using spare capacity on the local SSD as a cache for cloud backed data.

You have already had hands-on experience with AWS, and experienced how cloud storage can be a commercial service that its users get billed for using. The incurred cost can be broken down into three parts: capacity used, number of operations executed, and amount of data transferred. Table 1 summarizes the cost of each operation in our CloudFS universe (our tests), and in Amazon S3.

Type	**Price used in our tests**	**Real price of Amazon S3**
Capacity	$ 0.03 per MB (our tests bill you for your max usage during each test)	$ 0.085 per GB per month for the first 1 TB
Operation pricing	$ 0.01 per request	PUT, COPY, POST, LIST: $ 0.005 per 1,000 requests. GET: $ 0.004 per 10,000 requests.
Data transfer pricing	$ 0.09 per MB (outgoing from S3 only)	$ 0.12 per GB for up to 10 TB (outgoing from S3 only)

Table 1: Cloud Storage cost model (Amazon S3 pricing: `http://aws.amazon.com/s3/pricing`)

The primary reason why we choose to store files in cloud storage is because we do not expect them to fit entirely in the local SSD storage that is available to CloudFS. Large files, however, are not always accessed in their entirety. And conveniently, we have already broken down large files in CloudFS into small manageable segments. Thus, if only a few segments of a file receive multiple accesses, it makes sense to cache them locally and reduce the transfer cost incurred when reading the data repeatedly from S3. As you can see in Table 1, the transfer cost per MB is 9x the cost of a single request, and 3x the cost of storing the data in S3.

2.1 Cache Characteristics

The cache you will be implementing as part of this checkpoint will be *persistent*, and the cached data will reside on the local SSD. As a result, **your cache policy must have *no* memory-based component**. Furthermore, you should continue to assume that large files may not fit entirely in memory, and may even exceed the size of the local SSD.

To simplify debugging and testing, you are expected to store all cached data under the same directory in your CloudFS namespace: `/.cache/`. This directory **should not be visible to the users of your CloudFS file system**. Another requirement is that your cache must be of fixed capacity. The size of the CloudFS cache

should be exposed as a command-line parameter. The expected command-line syntax for this parameter should be as follows:

```
./CloudFS [--cache-size <Kilobytes>] [--other-args]
```

We will read this command line parameter for you, and fill in the cache_size variable (in bytes) inside the cloudfs_state struct made available to you. It follows from this, that a cache size of 0 implicitly disables caching. We expect you to implement a **write-back cache, with contents that *persist* across mounts**. The implication of this is that you need to ensure that your snapshots capture the state of the file system correctly, without omitting any data.

2.2 Cache Replacement Policy

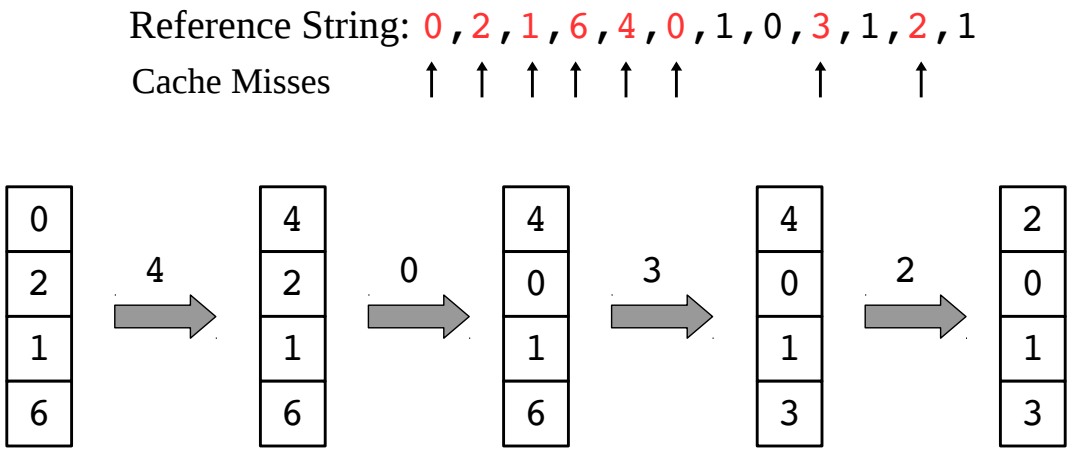

Figure 1: An example of the Least Recently Used (LRU) cache replacement policy.

Another important design decision is the **cache replacement policy**. Selecting and optimizing your cache replacement policy is a hard problem, so we request that you at least implement an eviction policy as simple as **Least Recently Used (LRU)**. This policy evicts the block of data (in your case: data segment) that was referenced least recently. An example is shown in Figure 1. When the cache is full and block 4 is referenced, block 0 is removed as the least recently referenced. The reference string shown incurs 8 cache misses out of 12 block references, so the cache's miss rate is estimated at 67%.

2.3 Grading Function

There are 17 test cases in total for Checkpoint 3 - 1-14 are for snapshot correctness, 15 is for cache performance, and 16-17 are for cache correctness. Test 15, the cache performance test consists of 7 workloads run back to back, and will output a total cost based on your cloud usage. Your goal is to minimize this total cost, with a total cost of $19 or less receiving full credit, and a total cost of $37 or more receiving no credit. The grading function for tests 15-17 is:

$$score_{cache} = min(max(\frac{37-YourTotalCost}{37-19}, 0), 1) * \frac{ScoreForTest16+ScoreForTest17}{2}$$

Note that if your total cost on test 15 is $37 or higher, you will receive no credit (0%) for tests 15-17.

Your autolab score on Checkpoint 3 will be determined 50% by tests 1-14, and 50% by $score_{cache}$. In other words, if your total cost on test 15 is $37 or higher, you will receive no higher than 50% on Checkpoint 3 autolab.

2.4 Requirements Summary

To summarize, the requirements for your cache design are the following:

- You are caching data at the granularity of Rabin segments.
- Cached data must be stored under `/.cache/` on the SSD. This directory must not be visible to users. If you're storing the cached data in any other place, you'll lose all score on cache test once being found out.
- Your cache must be write-back, and persistent across mounts.
- Your cache replacement policy must be at least as efficient as LRU [2].

3 Project Resources

The following resources are available in the project distribution that can be downloaded from Autolab.

- **CloudFS starter code**

  The files inside `src/` are the starter code that you will modify and extend for your project. `cloudfs.h` and `cloudfs.c` contain the starter code for your FUSE file system. `cloudapi.h` and `cloudapi.c` contain wrapper functions for the libs3 C library. The file `cloud-example.c` provides a usage example for enabling our libs3 wrapper to communicate with the Amazon S3-like server. The file `rabin-example.c` provides an example that demonstrates usage of the Rabin Segmentation API. Use the `make` command under `src/` to create the `cloudfs` executable.

  While this code is mostly the same as the distribution you received for Checkpoint 2, **we strongly encourage you to download the new version** which includes minor fixes and additional tests.

- **AWS instance deployment**

  Detailed instructions on the usage of AWS instances for CloudFS development are included in the distribution under `src/aws/README.aws`. The key necessary for accessing the instance is also available in the same directory, under the name `746-student.pem`.

- **AWS instance setup scripts**

  There are three scripts: `format_disks.sh`, `mount_disks.sh`, and `umount_disks.sh`, that are required to manage the AWS instance environment with the SSD. All the scripts are placed in the

[2] *More sophisticated policies will score higher in the scoreboard, though!*

`scripts/` directory and have a `README` file that describes their usage in details. Note that these scripts are provided for your assistance; it is a good idea to write your own scripts to debug and test your source code. Tools such as `gdb`, `blktrace`, and `valgrind` have already been installed in the provided AWS template.

- **Amazon S3 simulation web server**

  The file `src/s3-server/s3server.pyc` is compiled python code that simulates the Amazon S3-like cloud storage service. To run the web server, you can use the command: `python s3server.pyc`. The web server depends on the Tornado web server framework (`http://www.tornadoweb.org/`), which has been already installed in the provided AWS template. It stores all the files by default in `/tmp/s3/` (do not change this). To enable logging, you can simply run it with the `--verbose` option. You can see more options by using the `--help` option.

- **CloudFS snapshot API definitions and examples**

  The `src/snapshot/` directory contains two files: `snapshot-api.h` includes the necessary API definitions for the snapshot interface you will be implementing, and `snapshot-test.c` contains the parsing logic to transform command line arguments to `ioctl` operations to CloudFS. Examples for using the snapshots are provided in the `src/README` file. **Please run make inside the src/snapshot directory before running any tests.**

- **Data archiving**

  In order to send files to the cloud with reduced operation cost, it might be a good idea to package them as an archive (along with necessary metadata). You are not required to adopt this approach, but if you choose to do so you can select from a variety of libraries:

  - ▷ **`libtar`**. The standard archiving library. Note that it does **not** support extended attributes.
  - ▷ **`libarchive`**. Popular archiving library. If you choose to use it, you are required to stick with version 3.1.2, available at `http://libarchive.org/downloads/libarchive-3.1.2.zip`. Using a different version will invalidate your submission.
  - ▷ **`system`**. You could use the `system()` function to call `tar`, however this approach will result in forking additional processes. If you choose this approach, it will be your responsibility to debug your code and ensure that it works correctly. A file system that spawns a shall or a different process to perform internal work is a dangerous design decision.

  If back in checkpoint 1, you kept metadata information in extended attributes of a proxy file, we recommend you use the libarchive library, because libarchive supports reading and writing with the pax format. The pax format supports extended attributes, so if you use the `archive_write_set_format_pax()` command, libarchive will include extended attributes in the tar file. Some examples of using libarchive is available at `https://github.com/libarchive/libarchive/tree/master/examples`.

  Note that you are responsible for making sure that your method works on Autolab. If you would like to use a different archiving library, make sure to obtain our permission, and we will include it on this

list. Any libraries not listed will not be accepted as valid submissions.

4 Deliverables

The homework source code and project report will be graded based on the following criteria (subject to change).

- **Checkpoint 1:** Hybrid file system spanning SSD and cloud storage **25%** of grade
- **Checkpoint 2:** Block-level deduplication **25%** of grade
- **Checkpoint 3:** Snapshots and Caching **25%** of grade
- **Final project report:** Project report and source code documentation **25%** of grade

For each of the Checkpoint 1, 2, and 3 milestones you should submit a .tar file that contains only a directory `src/` with the source files of CloudFS. You should use the same code structure as in the archive you downloaded from Autolab, and make sure that there exists a Makefile that can generate the binary `src/build/bin/cloudfs`. We will test this in Autolab, and your code should compile correctly (Make sure to test this yourself!).

In your final project report submission you should submit a file `AndrewID.tar.gz`, where `AndrewID` is *your* Andrew ID, which should include at least the following items (and structure):

- A `src/` directory with any test suites you used for evaluation in your report. Please keep the size of test suite files small ($\leq$ 1MB), otherwise omit them. You will have already made your final code submission in Checkpoint 3, so **do not resubmit code here.**
- A `AndrewID.pdf` file containing your final project report, which should consist of **4 pages or fewer**. More information on the contents of this report follow.
- A `suggestions.txt` file with suggestions about this project, i.e., what you liked and disliked about the project and how we can improve it for the next offerings of this course.

4.1 Source code documentation

The `src/` directory should contain all your source code files. Each source code file should be well commented to highlight the key aspects of each function, while avoiding very long descriptions. This applies to each checkpoint submission you will make. Feel free to look at well-known open source projects' codebases to get an idea of how to structure and document your source code.

4.2 Final project report

The report should be a PDF file no longer than 4 pages in a single column, single-spaced 10-point Times Roman font with the name `AndrewID.pdf`. By now, you should know how annoying it can be to your instructors when you forget to present your Andrew ID front and center!

The report should contain design and evaluation of all three checkpoints, i.e., discuss Checkpoint 1, Checkpoint 2 and Checkpoint 3 goals, design, evaluation and evidence of success. The design should describe the

key data structures and design decisions that you made; often figures with good descriptions are helpful in describing a system. The evaluation section should describe the results from the test suite provided in the hand-out and your own test suite.

Your report must answer the following questions explicitly:

- Explain all the cost/performance trade-offs you encountered as part of this project
- Explain the trade-offs in choosing the right average segment size for deduplication
- Explain how deduplication is used in your design
- Explain your snapshot design, and the techniques you used to minimize cloud costs
- Explain your cache replacement policy, and how your policies translate into (hopefully) observed performance improvement and cost savings

4.3 Submissions

As always, you will submit your work through Autolab and will not be penalized for the first 25 submissions. Autolab test results will be used in your grade, but we will also perform additional tests that are not available to you. For the final submission, make sure to use the same directory structure provided in the code handout.

As a final reminder, make sure to **use the SSD for all CloudFS data that must be stored locally**. Specifically, you should not utilize locations outside your FUSE-mounted file system to store CloudFS data (or metadata). This excludes any information directed to `/tmp/cloudfs.log` for logging purposes.

4.4 Useful Pointers

- `https://github.com/libfuse/libfuse` is the de-facto source of information on FUSE. If you download the latest FUSE source code, there are examples included in the source. In addition, the FUSE documentation might prove helpful in understanding FUSE and its data structures:

  `http://libfuse.github.io/doxygen/`

  You can search Google for tutorials on FUSE programming. Some useful tutorials can be found at:

  `http://www.ibm.com/developerworks/linux/library/l-fuse/`

  `http://www.cs.nmsu.edu/pfeiffer/fuse-tutorial/`

- Disk IO stats are measured using `vmstat -d`. More information on it can be found using the man pages. `btrace` and `blktrace` are useful tools for tracing block level IO on any device. Read their man pages to learn about using these tools and interpreting their output.

- We have provided instructions on Amazon S3 API specifications. More information about Amazon S3 API specifications can be found at the following URLs:

  `http://libs3.ischo.com.s3.amazonaws.com/index.html`

  `http://aws.amazon.com/s3/`