Google Summer of Code

THE APACHE® SOFTWARE FOUNDATION

Support Database Backup to Cloud Storage

Kvrocks

Table of Contents:

1. Introduction

My name is **Anirudh Lakhanpal**, a 2nd-year Computer Science student at Graphic Era Deemed To Be University with a strong foundation in C++, having coded in it since high school. I am well-versed in modern C++ (C++11/14/17/20), data structures, algorithms, multithreading, and system programming.

I have hands-on experience working with databases (RocksDB, Redis) and cloud platforms (GCP, AWS). Additionally, I hold relevant certifications in cloud computing and databases.

Alongside my technical experience, I'm highly responsive and proactive in communication. I consistently maintain quick feedback loops with mentors and collaborators to keep progress smooth and transparent.

I am highly engaged in open-source development and have contributed to multiple large scale projects, including **NeutralinoJS, CGAL,** and **GTest**. Beyond contributing, I have also mentored multiple projects in various open-source programs, helping others navigate the ecosystem and build impactful software.

Contributions made to **Kvrocks** project**:**

- https://github.com/apache/kvrocks/pull/2826
- https://github.com/apache/kvrocks/pull/2822
- https://github.com/apache/kvrocks/pull/2849
- https://github.com/apache/kvrocks-website/pull/282
- https://github.com/apache/kvrocks-website/pull/276
- https://github.com/apache/kvrocks-website/pull/285
- https://github.com/apache/kvrocks-controller/pull/288
- https://github.com/apache/kvrocks-controller/pull/274#issuecomment-2720052162
- https://github.com/apache/kvrocks/pull/2832#issuecomment-2729845808

LinkedIn: https://www.linkedin.com/in/lanirudh

GitHub: https://github.com/SharonIV0x86/

2. Problem Statement

This project aims to implement a robust backup system that allows users to store Kvrocks backups directly in cloud storage services such as Amazon S3, Google Cloud Storage, and/or Azure Blob Storage. The solution will integrate with the existing Kvrocks backup and restore mechanisms while ensuring efficient and secure data transfer.

Deliverables:

1. **Cloud Storage Integration:** Implement backup storage support for Amazon S3, Google Cloud Storage, and Azure Blob Storage using SDKs, REST APIs or libraries (e.g. Apache OpenDAL).
2. **Backup & Restore Commands:** Extend Kvrocks' backup functionality to allow exporting and importing database snapshots from cloud storage.
3. **Configuration & Authentication:** Provide user-configurable options to specify storage credentials and backup parameters.
4. **Incremental Backup Support (Stretch Goal):** Optimize storage usage by implementing differential or incremental backup capabilities.
5. **Documentation & Tests:** Comprehensive documentation and test coverage to ensure reliability and ease of use.

Difficulty: Major
Project size: ~350 hour (large)
GSoC Idea Link

2.1 Benefits to the Community

Implementing a cloud backup and restore system for **Kvrocks** will significantly enhance its reliability, scalability, and usability for production deployments. Users will gain the ability to securely store and restore database snapshots from cloud storage, reducing the risk of data loss and simplifying disaster recovery. This feature will also enable seamless data migration across different environments and infrastructure setups. Additionally, by providing a well-documented and tested solution, this project will lower the barrier for adoption, making Kvrocks a more competitive alternative to other Redis-compatible databases in cloud-native applications.

3. Understanding The Backup and Restoration System

3.1 **Kvrocks** + **RocksDB** Backup Mechanism

The backup process for **Kvrocks** leverages RocksDB's checkpoint mechanism, which efficiently creates a consistent snapshot of the database by hard-linking immutable SST files and copying necessary metadata files. The **CheckpointImpl::CreateCheckpoint** function in RocksDB initiates the backup process by first verifying the existence of the target directory and cleaning up any temporary files. It then disables file deletions temporarily to ensure consistency before invoking **CreateCustomCheckpoint** function. This function iterates through all live files in the database, checking if they can be hard-linked (if they reside on the same filesystem) or need to be copied (if linking is not possible). Additionally, for files like **CURRENT**, which contain metadata, the function directly writes their contents to the checkpoint directory.

The `CreateCheckpoint` function in RocksDB and the `CreateBackup` function in Kvrocks both aim to create a consistent snapshot of the database by working together, but they differ in approach, RocksDB's checkpointing relies on linking or copying live SST files, whereas Kvrocks' backup mechanism ensures data integrity by managing file deletions and temporary directories. The process begins by checking if the target backup directory exists using `db_->GetEnv()->FileExists(checkpoint_dir)`, returning an error if it does. If not, a temporary backup directory is created and cleaned up if it contains any residual files from a previous attempt using `CleanStagingDirectory(full_private_path, db_options.info_log.get())`. The backup process then disables file deletions with `db_->DisableFileDeletions()`, ensuring that the files remain consistent throughout the operation. The core of the checkpointing logic is handled inside `CreateCustomCheckpoint`, where three types of file operations are performed: hard-linking files using `db_->GetFileSystem()->LinkFile(src_dirname + "/" + fname, full_private_path + "/" + fname)`, copying files if hard-linking is not possible with `CopyFile(db_->GetFileSystem(), src_dirname + "/" + fname, temperature, full_private_path + "/" + fname, ...)`, and creating essential metadata files with `CreateFile(db_->GetFileSystem(), full_private_path + "/" + fname, contents, db_options.use_fsync)`. This ensures that all SST files and WAL files are preserved correctly. Once all files are successfully copied or linked, file deletions are re-enabled with `db_->EnableFileDeletions()`. After that, the temporary backup directory is atomically renamed to the final checkpoint directory with `db_->GetEnv()->RenameFile(full_private_path, checkpoint_dir)`, ensuring that the backup is only exposed after a successful operation. If a failure occurs at any point, `CleanStagingDirectory(full_private_path, db_options.info_log.get())` ensures that no partial backups remain. Finally, if the operation is successful, the latest

sequence number is stored in `sequence_number_ptr`, ensuring the backup maintains consistency with ongoing database operations. Kvrocks follows a similar approach but integrates RocksDB's backup API while handling backup mutexes (`std::lock_guard<std::mutex> lg(config_->backup_mu);`) and logging each step to ensure robustness.

3.2 How Kvrocks Will Process Backup Uploads

In **kvrocks**, once the backup is written, we extend this approach by integrating a processing layer to optimize, encrypt, and prepare the backup for cloud storage. Once the snapshot is created, a dedicated backup process reads the snapshot directory, compresses large files using an efficient lossless algorithm like (**Zstd**) and encrypts them using **AES-256** or **ChaCha20** for security. This ensures that even if a backup file is intercepted, its contents remain protected. After encryption, the backup is chunked into smaller parts for efficient upload and to avoid hitting cloud provider size limits. The Kvrocks backup system then interacts with cloud storage providers like **Amazon S3, Google Cloud Storage, or Azure Blob Storage** via their SDKs or **OpenDAL** bindings (if C++ support becomes available).

The basic approach to writing a backup involves storing the backup files in a cloud storage **bucket** provided by popular services such as Amazon S3 (AWS), Google Cloud Storage (GCS), or Azure Blob Storage. The backup can be organized within the bucket using a structured folder hierarchy and appropriate file naming, making it easy to manage, access, and restore when needed. The **bucket** name can be a predefined one or provided by the user in a config file.

The upload process follows a **parallelized approach**, where multiple encrypted chunks are uploaded concurrently to maximize throughput while ensuring checkpoint integrity is maintained. Kvrocks also maintains a **manifest file** that contains metadata such as **sequence numbers, checksums, and file mappings**, which helps verify backup integrity during recovery.
To restore a backup, the system downloads and decrypts the files, reconstructs the database structure, and loads the data back into RocksDB. Depending on whether WAL files are included in the snapshot or the DB is flushed prior to backup, Kvrocks can optionally leverage RocksDB’s WAL replay mechanism during restore to ensure a consistent state. This will be explored and evaluated during the implementation phase.

```
tree
.
├── 000010.log
├── CURRENT
├── MANIFEST-000011
└── OPTIONS-000013

1 directory, 4 files
```

The tree above represents the structure in the backup folder of Kvrocks. As there are no SST files, it likely means that the memtable is not yet full and RocksDB has not yet flushed data into SST files. Kvrocks provides an inbuilt way to handle this by setting the **flush_before_backup** flag in **CreateBackupOptions**. Enabling this ensures that all unflushed data in the memtable is written to SST files before taking a backup, making the backup more complete and reducing reliance on WAL files.

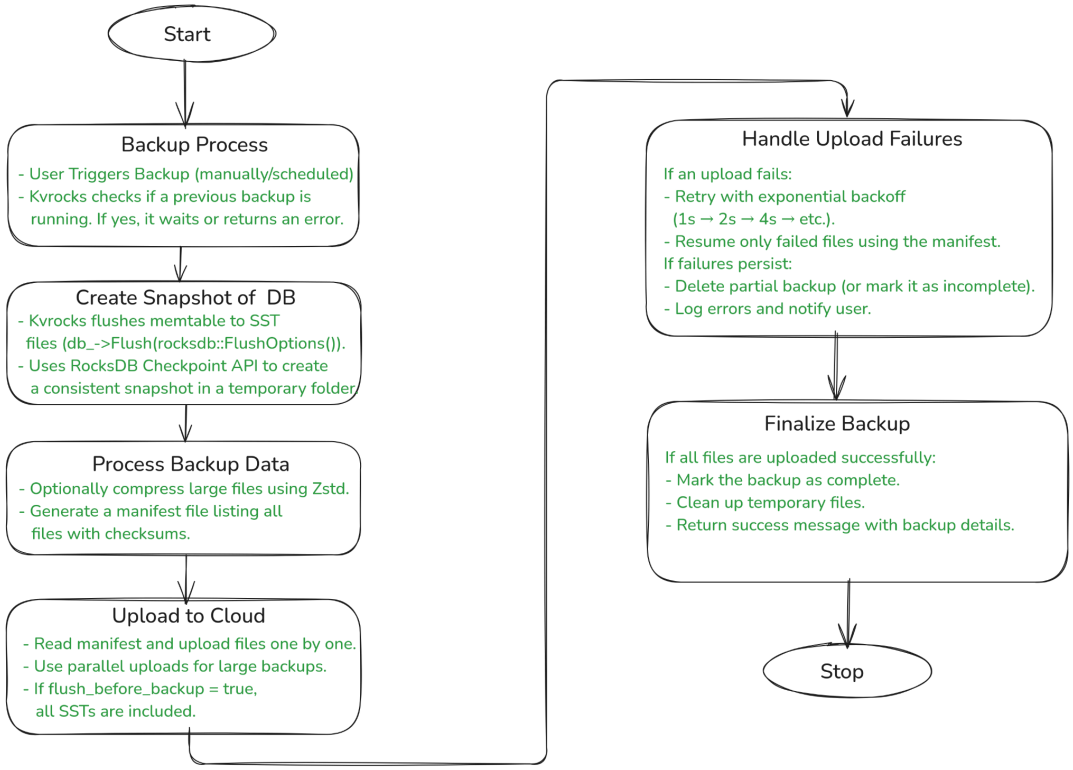
Additionally, if further control is needed, we can manually invoke RocksDB provided **db_->Flush(rocksdb::FlushOptions());** before triggering the backup process.

```
rocksdb::Status s = db_->Flush(rocksdb::FlushOptions());
if (!s.ok()) {
   LOG(ERROR) << "Failed to flush RocksDB memtable to SST. Error: " <<
s.ToString();
}
```

```
tree
.
├── 000035.sst
├── 000036.sst
├── 000037.log
├── CURRENT
├── MANIFEST-000038
└── OPTIONS-000040
1 directory, 6 files
```

The directory structure looks like this once the SST files are flushed.

The flowchart below demonstrates the whole backup process.

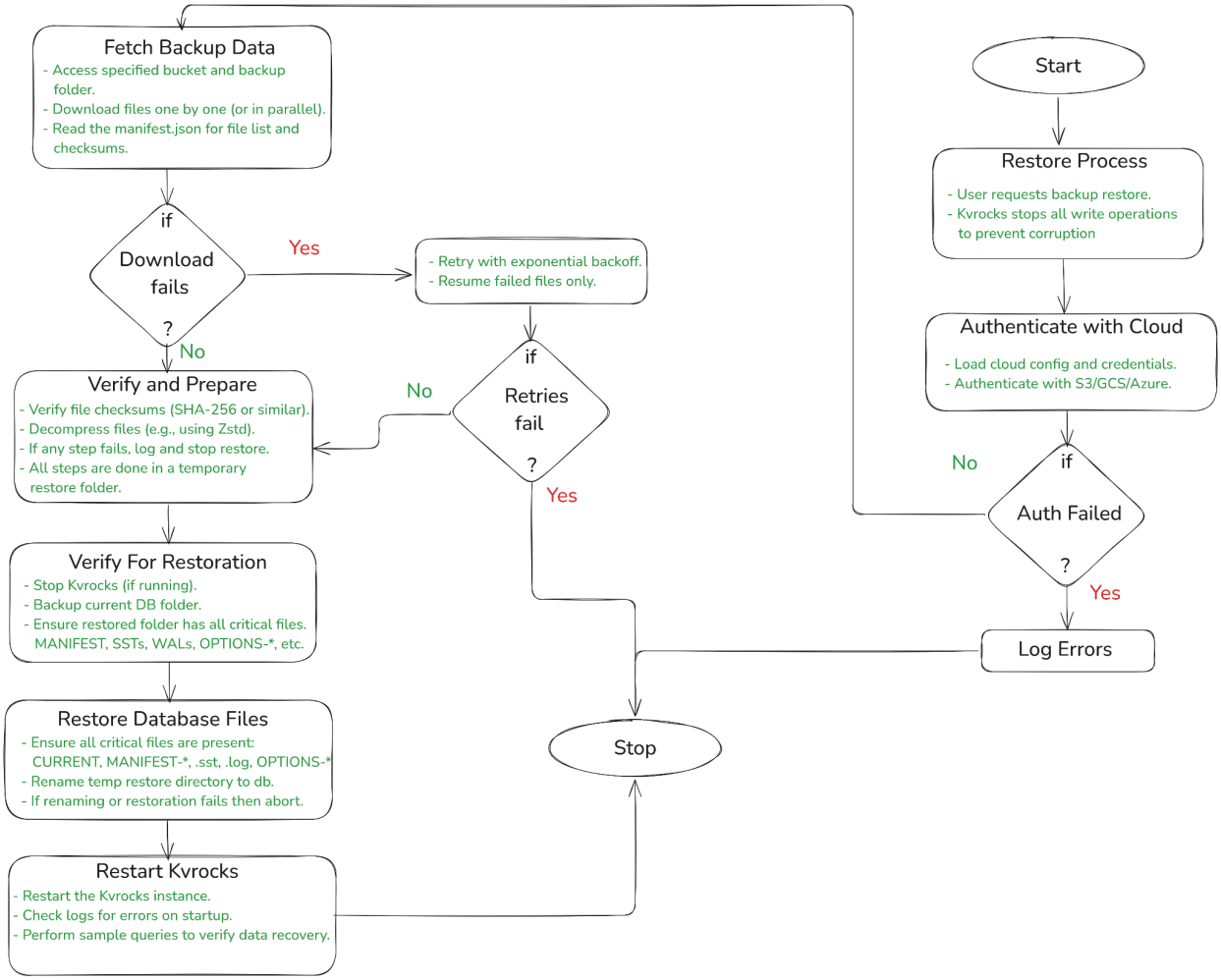

Start
Backup Process
- User Triggers Backup (manually/scheduled)
- Kvrocks checks if a previous backup is running. If yes, it waits or returns an error.
Create Snapshot of DB
- Kvrocks flushes memtable to SST files (db_->Flush(rocksdb::FlushOptions()).
- Uses RocksDB Checkpoint API to create a consistent snapshot in a temporary folder
Process Backup Data
- Optionally compress large files using Zstd.
- Generate a manifest file listing all files with checksums.
Upload to Cloud
- Read manifest and upload files one by one.
- Use parallel uploads for large backups.
- If flush_before_backup = true, all SSTs are included.
Handle Upload Failures
If an upload fails:
- Retry with exponential backoff (1s → 2s → 4s → etc.).
- Resume only failed files using the manifest.
If failures persist:
- Delete partial backup (or mark it as incomplete).
- Log errors and notify user.
Finalize Backup
If all files are uploaded successfully:
- Mark the backup as complete.
- Clean up temporary files.
- Return success message with backup details.
Stop

3.2 How Kvrocks Will Restore Backups

The restore system is designed with flexibility in mind, offering both a simple default workflow for quick recovery and an advanced approach for users who require finer control.

3.2.1. Simple Restore Workflow (Default):

The basic restore process begins with authenticating to the appropriate cloud provider. Once authenticated, the specified storage bucket is accessed, and the backup contents are fetched folder by folder and file by file. These files are then downloaded to a local directory, decrypted, decompressed, and restructured to match the original database layout. **Kvrocks** can then be restarted with this restored directory as its active database path, either by replacing the current DB or by loading from a user-specified location. This approach is straightforward and efficient for most users, enabling quick recovery with minimal steps. However, if **Kvrocks** is restarted using the restored directory by replacing the existing database path, the current database contents will be overwritten or deleted. To prevent data loss, users may optionally configure **Kvrocks** to restore the backup to a separate directory, ensuring the original instance remains intact for inspection or rollback.



3.2.2. Advanced Restore Workflow (Optional):

In scenarios where the user is already working with an active **Kvrocks** instance, we provide an extended restore workflow to avoid any data disruption. In this approach, the current database is preserved by moving it to a temporary backup location before initiating the restore process. Once the backup from the cloud is downloaded, validated, and structured locally, it can be plugged into **Kvrocks** as the active instance.

After restoration, the user has the flexibility to choose one of the following options:

- **Keep the restored instance** and discard the previous one.
- **Revert to the previous instance** by discarding the restored data.
- **Retain both instances**, where the restored instance runs as the primary, and the original is moved to a user-accessible directory for manual inspection or rollback. The path to the preserved instance is exposed to the user to simplify future access or restoration.

This workflow represents an advanced approach to restoration. However, when viewed from a broader perspective, it lays the foundation for a more comprehensive **database versioning system**, akin to version control systems like Git. Such a feature could allow users to seamlessly switch between multiple database states or "snapshots" over time. While out of scope for the initial implementation, this can be considered for future development.

4. Solution Approach 1

Using OpenDAL's Rust Core Binding.

OpenDAL, the **Open Data Access Layer**, is an Apache project designed to provide a unified interface for interacting with a wide range of storage services, including cloud platforms like AWS S3, GCP, and Azure Blob Storage, as well as other storage systems. It abstracts away the differences between these services, allowing developers to work with them seamlessly using a single API. OpenDAL is actively maintained and widely used, making it a reliable choice for implementing cloud backup functionality.

To integrate OpenDAL with **Kvrocks** for cloud backups, we can trigger the backup process from the user side and invoke OpenDAL's Rust core to handle the interaction with the specified cloud service. OpenDAL will manage the connection, authentication, and upload of the backup data to the user's chosen cloud storage. Since OpenDAL's C++ bindings are still a WIP and can take some time to mature even after its release, we can leverage its Rust core and expose it to Kvrocks via **FFI (Foreign Function Interface)**. This approach allows us to keep the communication between C++ (**Kvrocks**) and Rust (OpenDAL) minimal, ensuring a clean and efficient integration.

4.1 Developing Backup Commands for Kvrocks

This sub-section discusses the kvrocks backup commands for both the sub approaches listed in upcoming sub-sections **4.3(Implementing C++ to Rust communication via minimal FFI)** and **4.4(Implementing Rust to C++ via File Based Communication NO FFI)**.
With the backup system integrated into Kvrocks, users can trigger a backup upload to a cloud provider using the **KVCLOUD** command. This command ensures that **Kvrocks** securely transfers backup files to the configured cloud storage, based on user-defined settings in the **kvcloud.conf** file.

Command: **KVLOUD <ACTION>**

The action specifies the action to perform.

1. **UPLOAD -** Initiates the backup upload to the configured cloud provider.
    - Uses credentials and settings from **kvcloud.conf**
    - Will support AWS S3, GCS and Azblob.
    - Uploads the latest snapshot of **kvrocks** backup defined in the **backup_path** from the **kvcloud.conf**

2. **STATUS:** Retrieves the status of the last backup operation.
    - Returns success, failure, or in progress operation with timestamp in **ISO8601** format.
    - Provides error messages if an upload fails.

3. **RESTORE -** Fetches the kvrocks backup stored in the specified storage bucket/container.
4. **CONFIG:** Displays the currently loaded cloud backup configuration.
    - Lists all the parameters defined in the **kvcloud.conf**.
5. **REFRESH:** Reloads **kvcloud.conf** while the redis-cli connection is active, applying any configuration changes immediately without requiring a server restart.

kvrocks.conf structure

```
action = upload | restore
provider = aws
bucket = my-kvrocks-backup
region = us-east-1
access_key = <ACCESS_KEY>
secret_key = <SECRET_KEY>
backup_path = /tmp/db/kvrocks

action = restore
provider = gcs
bucket = my-kvrocks-backup-gcs
project_id = my-gcs-project
service_account_key = <SERVICE_ACCOUNT_KEY>
backup_path = /tmp/db/kvrocks
```

The **action** field in **kvcloud.conf** defines the allowed operations (upload or restore). If a cloud provider is listed in the configuration but does not have a specified action enabled, Kvrocks will not perform that operation for the provider.
If multiple providers are configured in **kvcloud.conf**, backups will be uploaded or restored to all of them by default. However users can specify a target provider explicitly using the CLI.

For Example: For GCS, only restore action is enabled in the config, so no uploads will be allowed to gcs.

```
KVCLOUD <UPLOAD | RESTORE > <AWS | GCS | AZ>
```

4.2 FFI Complexity

Foreign Function Interface can introduce an overhead complexity when trying to trigger or invoke Rust code from C++ code. Since we will be invoking rust code that uses OpenDAL from the C++ side.
We can simplify the given approach to minimize the communication between Rust and C++ code.
To do so we can follow the given steps.

1. Use OpenDAL Rust to handle the entire backup process (compression, encryption, and uploading to cloud storage).
2. From C++ (Kvrocks), only pass the necessary parameters (e.g., backup directory, target cloud storage URL etc) to the Rust code. Or we can also avoid passing parameters by writing all these details into a file at a specified path known to the rust code as well such as (**kvcloud.conf**), and rust code can use that information to work with the backup.
3. The Rust code will handle everything else and return a success/failure signal to C++.

4.3 Implementing C++ to Rust communication via minimal FFI.

From C++ (**Kvrocks**) we will pass the necessary parameters (eg: backup directory path, encryption choice, target cloud storage url, bucket name, etc. OR path to a file containing all this information) to the Rust code via FFI to keep the FFI interaction as minimal as possible.

Step 1:

Create a rust function that takes the backup directory, cloud storage url, region ,etc. as input strings.
From OpenDAL docs
Rust Implementation (lib.rs)

```rust
use std::ffi::{CStr, CString}; use std::os::raw::c_char;
use std::sync::Arc; use anyhow::Result;
use opendal::services::S3; use opendal::Operator;
use tokio::runtime::Runtime;
#[no_mangle]
pub extern "C" fn backup_to_s3(
    bucket: *const c_char,
    region: *const c_char,
    backup_path: *const c_char,
) -> bool {
    let bucket = unsafe { CStr::from_ptr(bucket).to_str().unwrap() };
    let region = unsafe { CStr::from_ptr(region).to_str().unwrap() };
```

```
    let backup_path = unsafe {
CStr::from_ptr(backup_path).to_str().unwrap() };
    let runtime = Runtime::new().unwrap();
    let result = runtime.block_on(upload_file_to_s3(bucket, region,
backup_path));
    result.is_ok()
}
async fn upload_file_to_s3(bucket: &str, region: &str, backup_path: &str)
-> Result<()> {
    let mut builder = S3::default()
        .bucket(bucket)
        .region(region)
        .endpoint("https://s3.amazonaws.com");
    let op: Operator = Operator::new(builder)?.finish();
    let data = std::fs::read(backup_path)?;
    let file_name = backup_path.split('/').last().unwrap();
    op.write(file_name, data).await?;
    Ok(())
}
```

This implementation demonstrates how Rust and C++ can interact to enable S3 backups for **Kvrocks**. The actual integration may require adjustments based on cloud provider requirements and Kvrocks' architecture.

The **backup_to_s3** function is called from the C++ code and converts the C-Style strings to native rust strings and then calls the **upload_file_to_s3** function which accesses the backup created by **Kvrocks** using **RocksDB** on the path specified by the **backup_path** variable. The code then proceeds to upload files to the object storage.

This approach can also be extended for backup restoration, fetching files from the cloud and storing them in **Kvrocks** backup directory. Errors such as network failures or authentication issues are handled in Rust before returning a status to C++.

Step 2:

Generating a C-Compatible Header using [cbindgen](). Cbindgen creates C/C++11 headers for Rust libraries which expose a public C API. To allow our C++ code to call this function we need to generate a header file which will be included in the **Kvrocks** code.
To create the C header file corresponding to our rust code we need to define a **cbindgen.toml** configuration file which might look something like this.
cbindgen.toml

```
language = "C"
```

```
include_guard = "MY_RUST_LIB_H"
```

Generating the header file.

```
cbindgen --config cbindgen.toml --crate my_rust_lib --output my_rust_lib.h
```

After specifying the options in the **cbindgen.toml** file we can proceed to generate the corresponding header file named **my_rust_lib.h** by executing the above command and passing the **cbindgen.toml** config file as a param.

Generated my_rust_lib.h

```
#ifndef MY_RUST_LIB_H
#define MY_RUST_LIB_H
#ifdef __cplusplus
extern "C" {
#endif
bool backup_to_s3(const char* bucket, const char* region, const char*
file_path);
#ifdef __cplusplus
}
#endif
#endif // MY_RUST_LIB_H
```

Above we have the C++11 compatible header file with the extern C code block specifying the function name and its parameter as written in the Rust code.
While this is doable by hand, it's not particularly good use of our time, and it's much more likely to be error prone than machine-generated headers that are based on the actual rust code.

Step 3:

Once we have the generated header files we can call the **backup_to_s3** function from our header file **my_rust_lib.h** generated by **cbindgen** from our C++ code (**Kvrocks**).
backup.cpp

```
#include "my_rust_lib.h"
#include <iostream>
#include <string>
void initiate_backup(const std::string& bucket, const std::string& region,
const std::string& file_path) {
   bool success = backup_to_s3(bucket.c_str(), region.c_str(),
file_path.c_str());
   if (success) {
       std::cout << "Backup to S3 successful!" << std::endl;
```

```
    } else {
        std::cerr << "Backup to S3 failed!" << std::endl;
    }
}
int main() {
    std::string bucket = "test-bucket";
    std::string region = "us-east-1";
    std::string file_path = "/path/to/local/file.txt";
    initiate_backup(bucket, region, file_path);
    return 0;
}
```

Above is a small prototype and can vaguely differ when integration with **Kvrocks** but the logic remains the same, from the **Kvrocks** side we have to call the appropriate function in this case **backup_to_s3** with appropriate parameters so that our Rust code can use OpenDAL to upload the backup to the specified object storage of the specified cloud platform.

Step 4:

Building and linking the rust library with **Kvrocks**. We need to compile the rust code as a shared library using the below command.

```
cargo build --release --crate-type cdylib
```

This will generate

- Linux/MacOS: **libmy_rust_lib.so**
- Windows: **my_rust_lib.dll**

Once the shared library is built, we need to compile the Kvrocks C++ code and link it with Rust. Assuming our C++ backup implementation is in **backup.cpp** we compile it as follows:

```
g++ backup.cpp -o backup -L/path/to/rust/target/release -lmy_rust_lib
```

Now, another potential issue is managing shared libraries. One way to handle this is by generating the required Rust shared libraries (**.so/.dll**) at build time, ensuring they are always up to date. Alternatively, we can ship pre-built shared libraries alongside Kvrocks, simplifying deployment but requiring us to maintain platform-specific binaries. Or there can be a possible workaround for this instead of the shared libraries.

4.3.1 Development Timeline

Development timeline for sub-approach **4.3**

Community Bonding Period (May 8 - June 1)

- Get to know about mentors and other Apache PMC members.
- Get familiar with Kvrocks' backup system (RocksDB checkpoints).
- Research FFI communication between C++ and Rust.
- Set up the development environment and create an initial project structure.

June 2 - June 14 (Week 1-2): Designing and Learning

- Learn Rust basics and prototype with OpenDAL.
- Design the backup system architecture (Rust, C++, RocksBD, OpenDAL).
- Discuss project design with mentors and finalize design decisions

June 15 - June 28 (Week 3-4): Establishing C++ to Rust Communication

- Write simple Rust-to-C++ FFI function calls for basic interaction.
- Use **cbindgen** to generate C-compatible header files.
- Implement the **backup_to_s3** function in Rust.
- Test basic C++ to Rust function calls with dummy parameters.

June 29 - July 12 (Week 5-6): Core Backup Implementation

- Explore the Rust backup function via FFI and integrate with **Kvrocks.**
- Incorporate use of **kvcloud.conf** to read credentials for authentication.
- Implement basic kvrocks backup commands **KVCLOUD UPLOAD | RESTORE.**
- Modify kvrocks backup mechanism to trigger cloud uploads.
- Implement basic error handling and logging for backup failures.

July 14 - July 18 (Midterm Evaluation Week)

Deliverables:

- Working Rust-based backup system integrated with Kvrocks.
- Support for AWS S3 and at least one other cloud provider (e.g., GCS).
- Demonstration of backup trigger from **Kvrocks**.
- Midterm progress report submission.

Coding Phase 2: Enhancements & Restoration

July 19 - August 4 (Week 7-8): Implementing Restore Functionality

- Implement Rust functionality to fetch backups from cloud storage.
- Develop a restoration mechanism to place files in Kvrocks' backup directory.
- Extend C++ logic to trigger restoration via FFI.
- Adding support for backups and restoration to Azure Blob Storage.

August 5 - August 18 (Week 9-10): Optimization & Robustness

- Optimize Rust's async execution using **tokio** for non-blocking backups.
- Implement a retry mechanism for failed uploads due to network failures.
- Improve logging and monitoring for backup success/failure tracking.
- Conduct performance benchmarking for different backup sizes.

August 19 - August 25 (Week 11): Final Testing & Documentation

- Write comprehensive documentation for backup and restoration usage.
- Conduct end-to-end testing across multiple cloud providers and add go tests.
- Ensure cross-platform compatibility (Linux, macOS, Windows).
- Finalize **Kvrocks** configuration options for cloud backups.

August 25 - September 1: Final Submission

Deliverables:

- Fully functional cloud backup and restore feature for **Kvrocks**.
- Well-documented Rust and C++ integration with OpenDAL.
- Performance benchmarks and test reports.
- Final GSoC submission, including code, documentation, and evaluation.

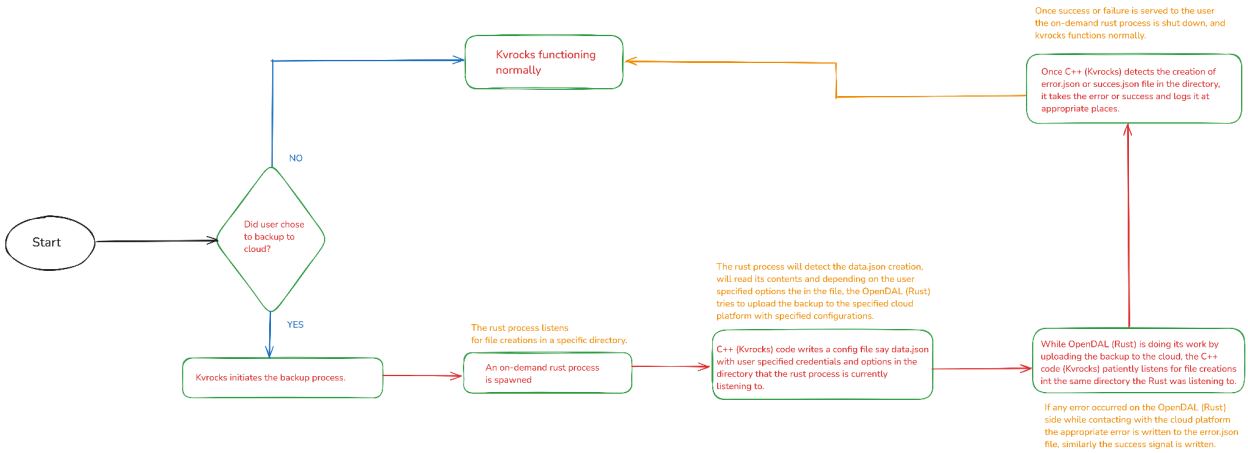
Optional Stretched Goal: September - November

- Research and implement **incremental and differential backups** for optimized storage.
- Refine the implementation based on mentor feedback.
- Submit PRs to upstream Kvrocks for official inclusion.

4.4 Implementing Rust to C++ via File Based Communication. (NO FFI)

This approach, not entirely straightforward but can prove to be very effective, and completely avoids the overhead of the FFI and shared libraries. This approach ensures that Rust only runs when necessary, eliminating FFI complexity and background processes while keeping **Kvrocks** performant. It leverages **file-based** communication and **on-demand Rust execution** for handling backups efficiently. This can also be implemented as a standalone tool as well.

The diagram below explains the whole process. (Zoom in)



Step 1:

While building **Kvrocks** we need to make sure to also build the Rust code which uses OpenDAL to communicate with cloud services. So we need to make sure that the Rust backup binary is compiled alongside **Kvrocks.** This step involves modifying the **Kvrocks** build system.

Step 2:

The **Kvrocks** should only trigger the spawning of the Rust process when a backup is needed or requested by the user. We need to add the Rust build commands in **Kvrocks'** build scripts (e.g: CMakeLists.txt).

```
cargo build --release
cp target/release/kvrocks-backup /build/kvrocks/bin/
```

Step 3:

After step 2, C++ created a specified backup directory, path of this backup directory must be already hard coded or available to the Rust code, because the rust binary from the spawned process will be listening for file creations or modification in this directory.

```
std::string backupDir = "/tmp/kvrocks-backup-com-dir";
fs::create_directories(backupDir);
std::cout << "Backup directory created: " << backupDir << std::endl;
```

The above code creates the directory in which both C++ (**Kvrocks**) and Rust (**OpenDAL**) will listen for changes.

Step 4:

Once the directory has been created, it's the time to spawn the Rust process.

```
std::string command = "/build/bin/kvrocks/kvrocks-backup " + backupDir + "
&";
   int status = std::system(command.c_str());
```

The above code launches the Rust binary as a background process, but this approach is specific to Unix-based systems. On Windows or other operating systems, a similar logic can be implemented with minimal adjustments.

After this we can check if the spawning process succeeded or not and can proceed accordingly. If the process spawned with no issues we can sleep for 0.5 seconds to give rust process some time to start up and listen for directory changes.

```
std::this_thread::sleep_for(std::chrono::seconds(2));
```

Step 5:

After all this, finally we will write the user-specified choices for cloud backup to the **data.json** file in the **kvrocks-backup-com-dir** in the json format, however a more space efficient file format can also be used later on and can be integrated with minimal adjustments.

Once the file is created, the Rust code detects this change and reads the content from the **data.json** and in the meantime the C++ **Kvrocks** code starts to listen for file creations or changes in the same directory while Rust (OpenDAL) does it works putting backup to the cloud.

Success or failure is indicated by the creation of the **success.json** or **failure.json** in the same directory and once C++ **Kvrocks** code detects this change depending on a success or a failure it takes further steps.

Step 6:

Testing.
For comprehensive testing of the cloud backup system in **Kvrocks**, I plan to implement tests in both **C++ and Go** to ensure testing at different levels. C++ tests will focus on validating lower-level components such as directory creation, process management, inter-process communication through file-based signaling, and success/error detection mechanisms. These tests will help ensure that the Rust backup process is correctly spawned, listens for file modifications, and properly communicates back to the C++ side.

On the other hand, Go tests will be used for **integration testing**, allowing us to interact with **Kvrocks** via the **RESP** protocol to simulate real-world scenarios. These tests will validate whether triggering a backup through **Kvrocks** results in a successful cloud upload and confirm that error handling mechanisms work as expected. By combining C++ unit tests for internal components and Go integration tests for end-to-end functionality, this approach will provide a reliable testing framework.

Step 7:

Documentation.
Throughout the development process, I will maintain well-structured markdown documentation to ensure seamless integration into the Kvrocks website later on. Additionally a proper **event logging** mechanism will be implemented to store the logs.

4.4.1 Development Timeline

Development timeline for sub-approach **4.4.**

Community Bonding Period (May 8 - June 1)

- Get to know mentors and Apache PMC members.
- Study Kvrocks' backup system (RocksDB checkpoints).
- Research OpenDAL and Rust for cloud storage integration.
- Discuss project details with mentors and finalize architecture.
- Set up the development environment and project structure.

Phase 1: Research and Initial Prototyping (June)

June 2 - June 14 (Week 1-2): Learning and Design Discussions

- Learn Rust and explore OpenDAL usage.
- Understand RocksDB checkpoint-based backups.
- Brainstorm user-configurable backup options (encryption, multipart, compression).
- Discuss Rust-to-C++ communication approaches with mentors.

June 15 - June 28 (Week 3-4): Rust Process Prototyping.

- Prototype spawning a Rust process from C++.
- Write a Rust program that listens for file changes.
- Test small-scale Rust-to-C++ communication using JSON files.
- Document findings for July's full implementation.

Phase 2: Full Development (July - August)

June 29 - July 12 (Week 5-6): C++ to Rust via File-Based Communication

- Implement robust **file-based communication** between Rust and C++.
- Modify Kvrocks' build system to compile Rust code alongside C++.
- Developing **backup_to_s3** and **backup_to_gcs** functions in rust to upload backups.
- Implement basic Rust process handling in C++ (start/stop Rust backup process).
- Develop basic backup commands in **Kvrocks** (e.g **KVCLOUD BACKUP**).

July 14 - July 18 (Midterm Evaluation Week)

- Implement Rust logic to **fetch backups** from cloud storage.

- Develop **restoration mechanism** (place files in Kvrocks' backup directory).
- Extend Kvrocks C++ logic to trigger restores via file-based signaling.
- Write Go integration tests for cloud restores.

Phase 3: Enhancements and Restore Feature

July 19 - August 4 (Week 7-8): Restore Functionality

- Implement Rust logic to **fetch backups** from cloud storage.
- Implement restore functionality from GCS and S3.
- Implement functionality for backup and restore to Azure Blob Storage.
- Develop **restoration mechanism** (place files in **Kvrocks'** backup directory).
- Extend **Kvrocks** C++ logic to trigger restores via file-based signaling.
- Write Go integration tests for cloud restores.

August 5 - August 18 (Week 9-10): Optimizations and Robustness

- Optimize Rust's async execution using **tokio** for non-blocking backups.
- Implement a **retry mechanism** for failed uploads.
- Add support for **encryption, multipart uploads, and compression** etc.
- Improve logging and monitoring for backup/restore events.

Final Weeks: Testing and Submission

August 19 - August 25 (Week 11): Final Testing and Documentation

- Write full documentation for Kvrocks Cloud backups.
- Conduct **end-to-end tests** across AWS, GCS and Azblob.
- Ensure cross-platform compatibility (Linux/macOS/Windows).

August 25 - September 1: Final Submission

- Fully functional **Kvrocks Cloud Backup & Restore system**.
- Complete documentation for Rust and C++ integration.
- Performance benchmarks and test reports.

Optional Stretch Goals (September - November)

- Research and implement **incremental/differential backups**.
- Improve performance based on real-world testing.

- Submit final PRs to upstream Kvrocks for official inclusion.

4.5 Conclusion

This approach leverages Apache's own software while also benefiting **OpenDAL** in terms of user base, making it a well-integrated solution. However, the added complexity of Rust introduces a learning curve, and the overall implementation is not entirely straightforward.

5. Solution Approach 2

Standalone Tool Using Native SDKs

This approach is straightforward, efficient, and highly flexible and great if we want to implement the backup logic in C++ directly into the existing **kvrocks** codebase or develop a standalone tool.
Instead of embedding cloud backup logic directly into **Kvrocks**, we can develop a standalone tool that leverages the native C++ SDKs of major cloud providers like **AWS**, **GCS**, and **Azure**. This tool will interact with **RocksDB** backups and handle the entire upload process, giving us full control over implementation while keeping **Kvrocks** core functionality untouched. This keeps the system clean, modular, and easier to maintain while still making full use of the capabilities provided by cloud providers. These SDKs are actively maintained by the cloud platforms. However this might add a bit of a dependency issue as while building **Kvrocks** we also have to build our backup tool which uses all three SDKs.

Google Cloud Storage C++ SDK.

- Docs
- Github

AWS C++ SDK

- Docs
- Github

Microsoft Azure Storage SDK.
The Azure Storage C++ Client Library has received a deprecation notice and is no longer being maintained as of March 29, 2025. While it can still be used, it is not actively supported. However, the latest Azure SDK for C++ is actively maintained, continuously updated, and is the best fit for our use case.

By utilizing the native C++ SDKs provided by AWS, GCS, and Azure, the tool aligns perfectly with Kvrocks' tech stack, which is also written in C++. This ensures better performance and better integration.

The use of native SDKs guarantees access to the latest features and optimizations offered by the cloud providers, ensuring reliability and efficiency.

This section highlights the design and development plans of a separate standalone cli tool called **kvrocks-cloud** to be used by users/developers to upload and restore **Kvrocks** backups to cloud services.

Additionally, instead of **kvrocks-cloud** being a standalone tool written in C++ it can also be directly integrated into existing **kvrocks** codebase with minimal changes as an API.

5.1 kvrocks-cloud CLI Arguments.

The **kvrocks-cloud** will be a simple CLI tool which later on can be also integrated with a GUI, but CLI interface gives more power and better control.

The full verbose command would be used this way to perform backups to the cloud.

```
kvrocks-cloud backup --provider s3 --bucket-name mybucket --backup-dir
/tmp/kvrocks --s3-key "PUBLIC-ACCESS-KEY:AWS-SECRET-ACCESS-KEY"
```

Or a short and concise command equivalent to this can also be worked out.

```
kvrocks-cloud backup -p s3 -b mybucket -d /tmp/kvrocks --s3-key
"AKIAEXAMPLE:mysecretkey"
```

However, the above two ways of writing commands should not be mixed together, as it can cause confusion and extra overhead to code.

Below is the part by part breakdown of the **kvrocks-cloud** CLI command.

- **kvrocks-cloud:** The name of the compiled CLI binary.
- **backup:** This is the main action to be performed, **backup/restore.**
- **--provider** or **-p:** To indicate which cloud provider or platform to upload or restore backup to. Other options will include (-**s3**, **gcs** and **az**).
- **--bucket-name** or **-b:** Name of the bucket or the storage container.
- **--backup-dir** or **-d:** Path to the **Kvrocks** backup created by **RocksDB** also defined in **kvrocks.conf.**
- **--s3-key:** Authentication credentials corresponding to the cloud platform. For different platforms different CLI args must be used as shown in the upcoming sections.

We can also add some additional parameters as CLI args which will be optional.

- **--encrypt** or **-e:** Boolean, to indicate whether to encrypt backup or not, can use in house encryption or use cloud provided encryption.
- **--multipart** or **-m:** Boolean, to indicate whether to perform backup uploads in multi-parts or not. If the user did not provide a **true** value for this, then depending on the size of the backup we can perform multi-part uploads by ourselves. But if the user specifically passed a **false** value then no matter what the size is upload takes place usually.
- **--cred-file** or **-cf**: This flag to indicate that you'd like to specify a path to a file containing the credentials necessary for authentication with the cloud platform.

--cred-file or **-cf** argument will work for all three cloud platforms, as this will be a separate credentials file (.txt or .json) and not the **kvrocks.conf.**

Only one of the arguments either **-cf** or **--s3-key** should be provided at a time. If both are specified priority will be given to the **--s3-key** or to any other hard-coded credential.

These commands and arguments are just an overview for the **kvrocks-cloud** tool; more commands and options can be extended and modified as needed during its development.

The following CLI commands arguments will be used in the upcoming sections, **5.2, 5.3 and 5.4.**

5.2 Auth Methods for Taking Backup to AWS via kvrocks-cloud CLI Tool

AWS provides multiple authentication options, this section explores all the authentication options for AWS which can be integrated into the standalone CLI tool **kvrocks-cloud** using the AWS C++ SDK. All the commands mentioned in this section will be specifically for uploading and restoring of **Kvrocks, RocksDB** backup to AWS.

1. Passing credentials via CLI.

A simple method would be to pass the credentials to the **kvrocks-cloud** via the CLI, but this introduces a security risk, and should be avoided, however as a developer sometimes there are unavoidable circumstances so this option will also be available.

The following command will be used to pass credentials via the CLI argos.

```
kvrocks-cloud backup -p s3 -b mybucket -d /tmp/kvrocks --s3-key
"AKIAEXAMPLE:mysecretkey"
```

2. Using AWS Access Key and Secret Key.

A better and common method involved storing credentials as environment variables so our tool **kvrocks-cloud** automatically searches and picks up the credentials and uses them for authentication.
Considering we have the credentials stored in the environment variables.

```
export AWS_ACCESS_KEY_ID="AKIAEXAMPLE"
export AWS_SECRET_ACCESS_KEY="mysecretkey"
export AWS_REGION="us-east-1"
```

```
kvrocks-cloud backup -p s3 -b mybucket -d /tmp/kvrocks
```

The above command can be executed from the user side if credentials (hard-coded as args) or a path to a credentials file is not provided. Then the **kvrocks-cloud** tool will first look for credentials in the environment variables.

3. Using AWS Shared Credentials File.

This approach is a bit similar to using the **-cf** flag with the **kvrocks-cloud** CLI tool. Another secure method is to use AWS Shared Credentials file (**~/.aws/credentials**). The AWS shared credentials file is a text file that stores your AWS credentials securely on your local machine. It is usually and automatically used by AWS SDKs but our standalone CLI tool can also use this.

The below command when executed.

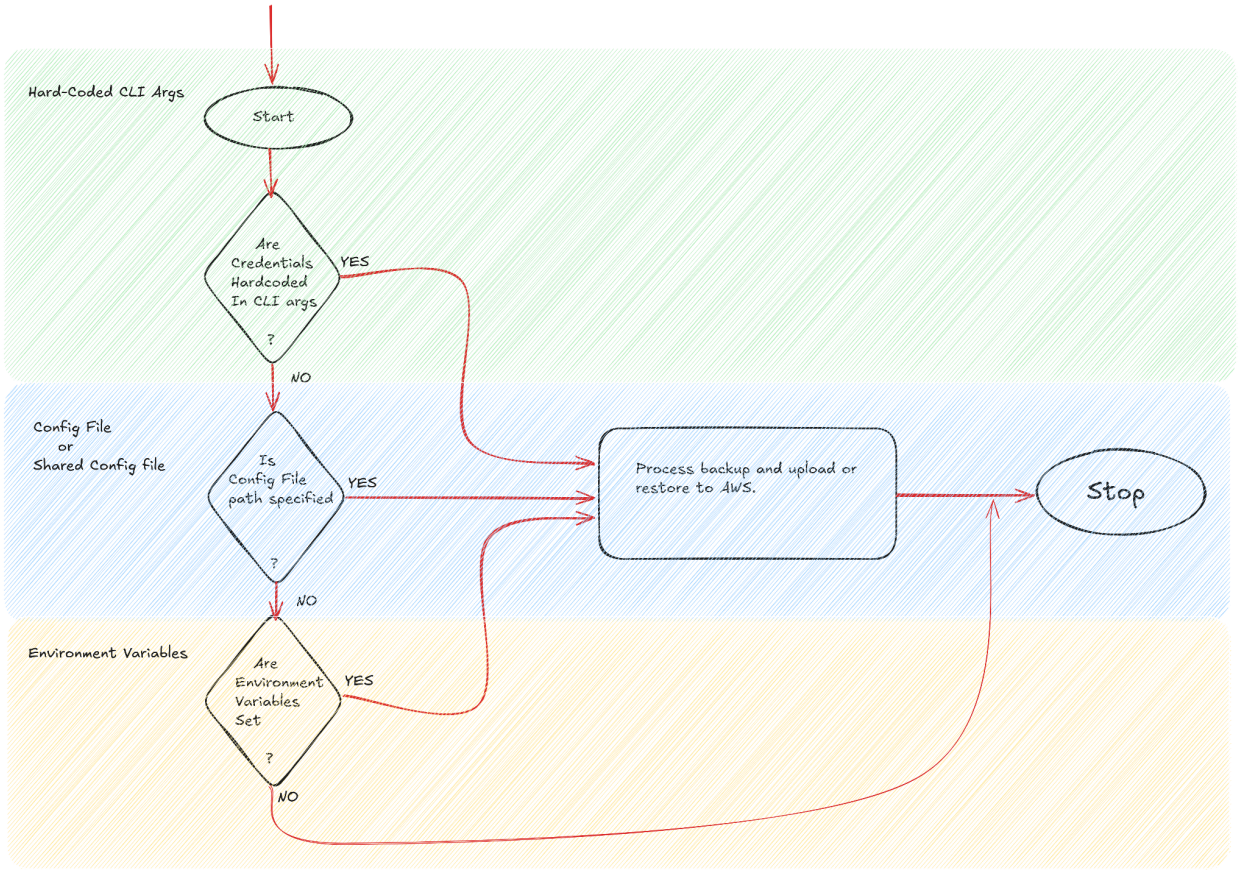
```
kvrocks-cloud backup -p s3 -b mybucket1 -d /tmp/kvrocks -scf
```

The **-scf** or **--shared-cred-file** is a boolean flag that will be automatically considered true when passed, this will indicate the **kvrocks-cloud** to search for the credentials stored in the Shared credentials file at **~/.aws/credentials**.

4. Using a Credentials File.

If a user/developer has their own credential file containing the information necessary for authentication they can also use it to perform backup upload and restore using the **--cred-file** argument or **-cf** argument to specify the path to this file, as mentioned in the sub-section **5.1 (kvrocks cloud CLI arguments).** The normal credentials file argument will be applicable and usable for all cloud platforms. However, a normal Credentials File and a Shared Credentials File are two different entities and are not the same.

The flowchart below explains the whole process and working of uploading or restoring backups from the AWS object storage.



Above diagram shows the highest priority will be given to the hard-coded credentials and then the configuration files and at the end to the environment variables. However, the priority order can be changed as needed.

5.3 Auth Methods for Taking Backup to Azure Blob Storage via kvrocks-cloud CLI Tool.

Azure provides two major authentication methods.

1. Passing credentials via CLI.

Simple. Easy. Fast. Not secure.

```
kvrocks-cloud backup -p az -b mybucket -d /tmp/kvrocks --azure-key
"myaccount:mysecretkey"
```

2. Using Storage Account Name and Access Key.

This approach is recommended. As there is no need to pass credentials via the cli.
In this approach we can avoid passing the credentials via the CLI and hint the **kvrocks-cloud** tool to first look for the following environment variables.

```
export AZURE_STORAGE_ACCOUNT="myaccount"
export AZURE_STORAGE_KEY="mysecretkey"
```

```
kvrocks-cloud backup -p az -b mybucket1 -d /tmp/kvrocks
```

If the above command is executed the **kvrocks-cloud** will first look for the environment variables to be set and if found will use them to authenticate and initiate the backup.

However, we can also specify a custom credentials file using **--cred-file** or **-cf** flag as mentioned in sub-section **4.1.**

From the SDK Docs.

```
std::string GetEndpointUrl() { return
std::getenv("AZURE_STORAGE_ACCOUNT_URL"); }
std::string GetAccountName() { return
std::getenv("AZURE_STORAGE_ACCOUNT_NAME"); }
std::string GetAccountKey() { return
std::getenv("AZURE_STORAGE_ACCOUNT_KEY"); }

auto sharedKeyCredential =
std::make_shared<StorageSharedKeyCredential>(GetAccountName(),GetAccountKe
y());
```

```
auto blockBlobClient = BlockBlobClient(GetEndPointUrl(),
sharedKeyCredential);
```

```
kvrocks-cloud backup -p az -b mybucket -d /tmp/kvrocks
```

The following command can be executed to upload or restore backups. Notice that if **kvrocks-cloud** is called with only the mandatory CLI arguments without explicitly providing credentials via a configuration file or hardcoded values-it will first attempt to retrieve authentication details from the environment variables based on the specified cloud provided by (**-p** or **--provider**) argument.

3. Using Azure SAS Token.

This method is the most secure, instead of exposing full storage keys, the user can generate a shared access signature Shared Access Signature (SAS) Token.
However the SAS TOken has limited permissions and expiry time.

Kvrocks-cloud can be called with following arguments to utilize SAS Token.
From one of the types of SAS tokens provided by azure, we can specifically utilize the a user delegation SAS is supported for Blob Storage and Data Lake Storage, and can be used for calls to **blob** endpoints and **dfs** endpoints.
Reference.

5.4 Auth Methods for Taking Backup to Google Cloud Storage via kvrocks-cloud CLI Tool.

Google Cloud Storage (GCS) offers multiple authentication methods, however most of them are for these services which are running inside the google cloud infrastructure. For foreign machines, the method below is the most suited and very versatile.

1. Service Account Key File.

This method is recommended for the machines which are trying to access GCP or GCS externally and not from inside the google cloud infrastructure.

In GCP, a service account is created which acts like a bot user with specific permissions. The private key JSON file associated with the account is used for authentication. The AWS equivalent of this method is IAM User Access Keys (point 2 in section **5.2(Auth Methods for Taking Backup to AWS via kvrocks-cloud CLI Tool)**

Below are the steps for creating a service account usingGCP.

Step 1:

First, the user/developer must create a service account from the Google Cloud console or using the **gcloud** CLI.

```
gcloud iam service-accounts create kvrocks-bkp-account --display-name
"Kvrocks Backup Service Account"
```

Step 2:

Grant the required permissions to the service account. The following command assigns the **Storage Admin** role (**roles/storage.admin**), which allows the account to manage GCS objects and buckets.

```
gcloud projects add-iam-policy-binding my-project-id \
--member="serviceAccount:kvrocks-backup@my-project-id.iam.gserviceaccount.
com" \
 --role="roles/storage.admin"
```

Step 3:

Generate the private key file (JSON) associated with the service account. This file is required for authentication

```
gcloud iam service-accounts keys create ~/kvrocks-backup.json \
 --iam-account=kvrocks-backup@my-project-id.iam.gserviceaccount.com
```

Step 4:

Setting the key file path. To authenticate using the generated key file, we need to set the **GOOGLE_APPLICATION_CREDENTIALS** environment variable.

```
export GOOGLE_APPLICATION_CREDENTIALS=~/kvrocks-backup.json
```

The **kvrocks-cloud** command to be executed will look like this, in order to authenticate using the Google Service Account

```
kvrocks-cloud -p gcs -b mybucketx -d /tmp/kvrocks -gsa
~/kvrocks-backup.json
```

Here the **-gsa** CLI arg indicates the path to the json file containing the application credentials. If **-gsa** is not passed then GCS C++ SDK employs ADC (Application Default Credentials) and automatically looks for the **GOOGLE_APPLICATION_CREDENTIALS** to be set.

In the code example below, the SDK automatically looks for the **GOOGLE_APPLICATION_CREDENTIALS** environment variable to be set and containing the path of the service account key file.

```
#include "google/cloud/storage/client.h"
#include <iostream>
#include <string>

int main(int argc, char* argv[]) {
 if (argc != 2) {
   std::cerr << "Missing bucket name.\n";
   std::cerr << "Usage: quickstart <bucket-name>\n";
   return 1;
 }
 std::string const bucket_name = argv[1];
```

```
  //  client to communicate with Google Cloud Storage. This client
  // uses the default configuration for authentication and project id.
  auto client = google::cloud::storage::Client();

  auto writer = client.WriteObject(bucket_name, "quickstart.txt");
  writer << "Hello World!";
  writer.Close();
  if (!writer.metadata()) {
    std::cerr << "Error creating object: " << writer.metadata().status()
              << "\n";
    return 1;
  }
  std::cout << "Successfully created object: " << *writer.metadata() <<
"\n";

  auto reader = client.ReadObject(bucket_name, "quickstart.txt");
  if (!reader) {
    std::cerr << "Error reading object: " << reader.status() << "\n";
    return 1;
  }

  std::string contents{std::istreambuf_iterator<char>{reader}, {}};
  std::cout << contents << "\n";

  return 0;
}
```

Another method is to explicitly provide the path to the service account key file.

```
#include <iostream>
#include <google/cloud/storage/client.h>

  int main() {
    try {
      auto client = google::cloud::storage::Client(

google::cloud::Options{}.set<google::cloud::UnifiedCredentialsOption>(
              google::cloud::MakeServiceAccountCredentialsFromFile(
```

```
                "~/kvrocks-backup.json"))); //Explicit Credentials

    auto buckets = client.ListBuckets();
    for(auto const& bucket : buckets){
        std::cout << bucket->name() << std::endl;
    }

  } catch (google::cloud::Status const& status) {
    std::cerr << "Error: " << status << "\n";
    return 1;
  }
  return 0;
}
```

For authenticating with Google Cloud Storage, particularly from external machines, we utilize service account key files. These JSON files contain the private key of a service account, a special Google Cloud identity for applications, granting it specific permissions. To streamline the authentication process, the Google Cloud C++ SDK employs **Application Default Credentials (ADC)**. ADC is a strategy that automatically locates credentials. If the **APPLICATION_DEFAULT_CREDENTIALS** environment variable is set, ADC uses the service account key file it points to. Otherwise, ADC checks well-known locations, like instance service accounts within Google Cloud environments. Thus, while a service account key file is a concrete credential, ADC is the mechanism that finds and uses it, offering flexibility by supporting multiple credential sources and simplifying authentication across diverse deployment scenarios.

5.5 Development Timeline

Development timeline for sub-approach **5.4(Auth Methods for Taking Backup to Google Cloud Storage via kvrocks-cloud CLI Tool).**

Community Bonding Period (May 8 - June 1)

- Get to know about mentors and other Apache PMC members.
- Study Kvrocks' backup system (RocksDB checkpoints) in depth.
- Research native C++ SDKs for AWS, GCS, and Azure
- Discuss project details with mentors and finalize the architecture for the standalone tool.

Phase 1: Research and Initial Prototyping (June)

June 2 - June 14 (Week 1-2): Learning and Design Discussions

- Study the native C++ SDKs for AWS, GCS and Azure and set up the SDKs.
- Understand the authentication mechanisms for each cloud provider (AWS IAM, GCP Service Accounts, Azure SAS Tokens).
- Brainstorm CLI design and user-configurable options (encryption, multipart uploads, compression).
- Discuss the integration of the standalone tool with Kvrocks' backup system.
- Document the design and architecture for the **kvrocks-cloud** tool.

June 15 - June 28 (Week 3-4): Prototyping and Initial Implementation

- Prototype basic CLI functionality (parsing arguments, handling commands).
- Implement a basic skeleton for the **kvrocks-cloud** tool.
- Test small-scale interactions with cloud SDKs (e.g., list buckets, upload a small file).
- Document findings and prepare for full implementation in July.
- Establish authentication mechanisms for AWS, GCS, and Azure (environment variables, credential files).

Phase 2: Full Development (July - August)

June 29 - July 12 (Week 5-6): Core Backup Functionality

- Implement the core backup functionality for AWS S3 using the AWS C++ SDK.
- Add basic support for GCS and Azure Blob Storage using their respective SDKs.
- Develop the logic to handle multipart uploads for large backups.
- Implement basic error handling and logging for backup operations.

- Extend the CLI to support all mandatory arguments (**--provider**, **–bucket-name**, **–backup-dir**, etc.).

July 14 - July 18 (Midterm Evaluation Week)

- Conduct midterm evaluation with mentors.
- Demonstrate basic backup functionality for cloud providers.
- Gather feedback and refine the implementation.

July 19 - August (Week 7-8): Restore Functionality

- Implement restore functionality for AWS S3, GCS, and Azure Blob Storage.
- Develop the logic to download backups from cloud storage and place them in the specified directory.
- Extend the CLI to support restore commands.
- Add support for optional arguments (**--encrypt**, **–multipart, --cred-file**).
- Adding more configuration options to **kvcloud.conf**
- Write integration tests for backup and restore operations.

Phase 3: Enhancements and Robustness (August)

August 5 - August 18 (Week 9-10): Optimizations and Additional Features

- Completing any leftover tasks from previous phases.
- Add support for encryption (in-house or cloud-provided).
- Improve logging and monitoring for backup/restore events.

August 19 - August 25 (Week 11): Final Testing and Documentation

- Write full documentation for the **kvrocks-cloud** tool (CLI usage, authentication methods, etc.).
- Ensure cross-platform compatibility (Linux/macOS/Windows).
- Fix any remaining bugs and polish the codebase.

Optional Stretch Goals (September - November)

- Fixing any bugs or issues that arise.
- Research and implement incremental/differential backups.
- Submit final PRs to upstream **Kvrocks** for official inclusion.

5.6 Developing a Standalone Tool in Python.

Instead of creating a standalone tool in C++ or integrating the same C++ logic into the existing **Kvrocks** codebase which uses C++ SDKs, a simple and lightweight tool in python can be developed to simplify the development process and provide the same options and configurations to the user as mentioned in sub-section **5.1(kvrocks cloud CLI arguments).** Major cloud providers such as (AWS, GCS, Azure) offers their python SDKs for the same. We can develop a standalone cli tool called **kvrocks-cloud** to upload backups to chosen cloud providers using python.
The SDKs and client libraries are available as follows.
AWS: https://aws.amazon.com/sdk-for-python/
GCS: https://cloud.google.com/python/docs/reference/storage/latest
Azure: https://learn.microsoft.com/en-us/azure/developer/python/sdk/azure-sdk-overview

The process and CLI arguments for the kvrocks-cloud tool will largely align with those outlined in Sub-section **5.1 (Kvrocks Cloud CLI Arguments)**. However, minor adjustments may be made based on specific requirements and feedback. Similarly, the development timeline for this tool is expected to follow the structure outlined in Sub-section **5.5 (Development Timeline)**, with possible refinements and minimal changes as needed.

5.7 Areas for Improvement & Considerations

Handling cloud SDK dependencies efficiently is crucial, as adding AWS, GCS, and Azure SDKs may introduce build dependency issues. To mitigate this, **kvrocks-cloud** can be distributed as a separate package (e.g., pre-built binaries or a Docker image) and use dynamic linking where possible to reduce build overhead. Credential management is another key aspect, as passing secrets via CLI (e.g., **--s3-key "AKIAEXAMPLE:mysecretkey"**) is insecure. Users will be warned against exposing secrets in CLI arguments, with environment variables and credential files recommended for authentication. Cross-platform support is also essential, ensuring **kvrocks-cloud** builds cleanly across Linux, macOS, and Windows.

Robust error handling and logging will be implemented, including structured logging (JSON format or current **kvrocks** logging mechanism) for better debugging and retry mechanisms like exponential backoff for failed uploads.

kvrocks-cloud could be implemented in another language like **Python** as mentioned in sub-section **5.6**, which would simplify development.

Another approach instead of separately launching the **kvrocks-cloud** tool would be to directly enter the **kvrocks-cloud** shell via the Redis CLI, when running the **Kvrocks** a command such as **KVCLOUD** can be executed to directly enter the shell of **kvrocks-cloud** tool and perform the backup or restoration and exit back to the Redis CLI session which was running before.

Additionally, while **kvrocks-cloud** is intended as an external tool, an alternative approach could involve integrating backup functionality directly into **Kvrocks**, enabling a Redis-like **BGSAVE CLOUD** command alongside the existing **BGSAVE** for seamless cloud backups. However, this would diverge from Kvrocks' goal of being a Redis-compatible database, as such commands do not exist in Redis.

Lastly, testing across multiple cloud providers is challenging due to the need for real accounts, so mock SDKs (e.g., LocalStack for AWS S3, MinIO, and Azurite for Azure) can be used, along with integration tests for real cloud environments in both C++ and go.

6. Availability and Commitments

The development timeline provided above outlines a general overview of the tasks and milestones planned for the duration of GSoC. While I will strive to follow this schedule closely, it is flexible and can be adapted as needed based on feedback, changing priorities, or unforeseen circumstances. I am committed to maintaining regular communication with my mentors and adjusting the plan when necessary to ensure consistent progress throughout the project.

I would also like to mention that I have my end-semester exams and practicals scheduled throughout the month of June, during which my availability will be limited. I plan to minimize the workload in June to focus on academics. However, I will ensure all major development tasks are scheduled before and after this period. Apart from June, I will be fully available and dedicated to the project, with significant time allocated for coding, testing, documentation, and communication.

7. Closing Words

A great deal of effort, time, and research has been put into crafting this proposal, not just to meet the requirements, but to truly understand the problem space and propose a practical, well-thought-out solution. This proposal, spanning nearly 40 pages, reflects my genuine enthusiasm, attention to detail, and dedication to the open-source community.

Throughout the proposal, I have explored and documented two possible solutions/approaches, each of which has been carefully evaluated for feasibility, performance, and maintainability. To the best of my knowledge and understanding, all proposed approaches are practical, implementable, and aligned with the project's goals. Personally, I'm inclined towards implementing the backup functionality using platform-specific SDKs by developing a standalone tool in Python or C++, as this approach offers flexibility, simplicity, and better separation of concerns. However, I'm completely open to exploring any other approach that the mentors or the wider community feel is more suitable for the project.

As highlighted in the problem statement, working with Azure Blob Storage or Azure Cloud Services in general is not as straightforward as working with AWS or GCP. Given that AWS and GCP dominate the cloud market in terms of adoption, ease of use, and developer experience, it is reasonable to consider the development of backup support for Azure Blob Storage as a lower-priority initiative. However, despite its complexities and the challenges associated with Azure's ecosystem, subsection **5.3 (Auth Methods for Taking Backup to Azure Blob Storage)** and **Approach 4 (Using OpenDAL's Rust Core Binding)** still provide valuable insights and functionality for enabling backups to Azure Blob Storage.

Google Summer of Code has been a dream opportunity for me, and I am wholeheartedly committed to making the most of it. If selected, I will give my 100% to ensure this project is a success, not just by delivering code, but by contributing meaningfully to the community, writing proper documentation, maintaining consistent communication, and learning continuously along the way.

Open source has already played a big role in shaping my journey as a developer, and GSoC will help me take that journey to the next level. Thank you for considering my proposal, I truly hope to have the opportunity to work on Apache **Kvrocks** and bring this idea to life.