TimeFuseDB: A Uni-Temporal Database using FUSE (Filesystem in User Space)

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Aniket Ray

May 2025

TimeFuseDB: A Uni-Temporal Database using FUSE (Filesystem in User Space)

THESIS

Submitted in Partial Fulfillment of

the Requirements for

the Degree of

MASTER OF SCIENCE (Computer Science)

at the

NEW YORK UNIVERSITY
TANDON SCHOOL OF ENGINEERING

by

Aniket Ray

May 2025

Approved:

_____

Department Chair Signature

_____

Date

University ID: N17957996
Net ID: ar8431

Approved by the Guidance Committee:

Major: Computer Science

Dr. Kamen Yotov
Adjunct Professor in Computer Science

Date

Vita

Aniket Ray, born on September 11, 1998, in *Asansol, West Bengal, India*, holds a Bachelor of Technology (B.Tech) degree in Computer Science and Engineering from the *National Institute of Technology (NIT), Durgapur*. Prior to commencing his graduate studies at *New York University (NYU)* in 2023, he gained significant professional experience with the *Mediapipe team* at *Google, Inc.*, and as a Software Engineer at *Oracle Corporation*. He is expected to complete his Master of Science degree in May 2025. The research presented in this thesis was conducted under the guidance of *Dr. Kamen Yotov* between January 2024 and March 2025.

ABSTRACT

TimeFuseDB: A Uni-Temporal Database using
FUSE (Filesystem in User Space)
by
Aniket Ray
Advisor: Assistant Prof. Dr. Kamen Yotov, Ph.D.
Submitted in Partial Fulfillment of the Requirements for
the Degree of Master of Science (Computer Science)

May 2025

This thesis presents `TimeFuseDB`, a file system architecture that integrates content-addressable storage with temporal versioning capabilities through a virtual file system interface. `TimeFuseDB` addresses the challenge of accessing historical versions of data while maintaining the familiar interface of a traditional file system. By employing efficient content hashing techniques and structured metadata storage, the system enables time-based querying of repository content via a regular file system interface.

Contents

List of Figures

Chapter 1

Introduction

Time is an essential dimension in understanding the evolution of data, yet traditional file systems and version control interfaces often obscure the temporal narrative of changes. This thesis introduces an innovative approach to representing temporal data by leveraging `Filesystem in Userspace (FUSE)` to expose the history of a `git` repository in an intuitive, navigable format. By integrating a custom crawler that traverses a `git` repository and extracts key metadata, this work constructs a virtual file system where each directory snapshot is labeled with a `Unix epoch` timestamp. This design enables users to "go back in time" seamlessly—simply by executing a command at the mounted directory such as '`cd TIMESTAMP-<UNIX_EPOCH_TIME>`'—to explore the state of the repository as it existed at any specific moment.

The novelty of this solution lies in its departure from conventional `git` interfaces. Traditional `git` operations depend on abstract commit logs and complex command sequences to reconstruct past states, which can be un-intuitive and cumbersome. In contrast, the proposed system offers a concrete, directory-based representation

of a repository's evolution. This extension of `git`'s capabilities not only simplifies temporal navigation but also provides a more accessible and tangible perspective on how data evolves over time, thereby enhancing tasks such as code auditing, financial analysis, and many other applications.

By re-imagining the navigation of historical data as a tangible, file system-based experience, this work bridges the gap between abstract version control operations and user-friendly temporal data exploration, significantly extending how `git` functions in practical, everyday use.

At the core of the system is a custom crawler designed to traverse a `git` repository and build a detailed representation of its commit history. The crawler processes each commit by extracting metadata and populating a `SQLite` database that serves as the backbone for mapping the temporal evolution of data. Instead of directly copying files from the repository, the system computes the `XXH128` hash for each file and stores them in a designated directory using a content-based addressing scheme. Custom object files are generated alongside; these objects act as pointers linking to other directories, thereby enabling the dynamic assembly of the file system structure.

The combined data—sourced from the `SQLite` database and the commit history—is then leveraged by `FUSE`. `FUSE` mounts a virtual file system at a specified location on the host system, dynamically reconstructing the repository's state as it existed at any given `Unix epoch` timestamp. In addition to allowing users to "go back in time," the system supports intuitive browsing of the commit history through a seamless, directory-based interface.

Chapter 2

Background

2.1 Temporal Data

Temporal data refers to information that evolves over time, capturing the history of changes to ensure both past and present states of the data are available. It serves as a critical aspect of modern data management, especially in domains where historical tracking and accurate timelines are essential. Temporal data can be categorized into various models based on the number of time dimensions it tracks. It is common to consider a single time dimension (uni-temporal), but having two time dimensions (bi-temporal) is not unusual, and in rare instances, more than two time dimensions are utilized.

2.1.1 Uni-temporal Data

Uni-temporal data involves tracking changes along a single time dimension. This time axis can represent either system time or valid time. System time records when changes are made in the database, offering a precise audit trail of

updates as they happen. Alternatively, the valid time represents the period during which the data is considered accurate in the real world context. By focusing on one timeline, uni-temporal data provides a simplified framework for maintaining historical records. It is particularly useful in scenarios where tracking corrections or maintaining multiple timelines is unnecessary. For example, in financial markets, uni-temporal models can monitor stock reference data—such as changes in the S&P 500 composition, adjustments to market capitalization, stock splits, and dividend announcements—by recording each event with a system timestamp, thereby creating a clear, chronological audit trail without the complexity of multiple time dimensions.

The simplicity of uni-temporal data makes it easy to implement. Its scope is limited to scenarios where only one temporal perspective is needed. Systems that require the flexibility to correct past data or distinguish between system-recorded changes and real-world validity would find uni-temporal data insufficient.

2.1.2 Bi-temporal Data

Bi-temporal data expand the concept of uni-temporal data by incorporating one more time dimensions: system time and valid time. This dual timeline approach allows the system to distinguish between when a record was entered into the database and when it is valid in the real world. The additional dimension enables a richer and more flexible historical tracking mechanism, accommodating scenarios where corrections or backdated changes are common. For example, in a financial system, bi-temporal data can record corporate actions—such as dividend announcements, stock splits, or mergers—by tracking both the event's effective date in the market and the timestamp when it was recorded in the system, thereby ensuring precise historical analysis and robust audit trails for regulatory compliance.

The ability to correct past records without losing the original context is a key advantage of bi-temporal data. It supports the seamless integration of retrospective updates and provides a transparent view of "what was known when" and "what was true when." This makes it ideal for regulatory compliance, legal accountability, and complex data systems. However, the added complexity of managing two timelines requires more storage, processing power, and careful query design.

2.2 Content based addressing

Content-based addressing is a method of accessing and identifying data based on its content rather than its physical or logical location. In this approach, each piece of data is assigned a unique identifier derived directly from the data's content, often using a cryptographic hash-function like SHA-256. The resulting hash value acts as a fingerprint for the data, ensuring that any change in the content produces a new distinct identifier.

This technique is particularly useful in systems that prioritize data integrity, immutability, and efficient storage. Since the identifier depends solely on the content, any modifications to the data result in the creation of a new version with a new hash. This immutability is a key characteristic that ensures that the stored data remain unchanged unless explicitly replaced.

Content-based addressing is also highly effective for de-duplication. Identical content across multiple files or objects will generate the same hash, allowing systems to store a single instance of the data while referencing it multiple times. This approach is widely used in distributed systems and content-addressable storage (`CAS`) to enhance efficiency, ensure data consistency, and verify integrity by verifying

that the content matches its hash-based identifier.

We have successfully employed content-based addressing in our project. By leveraging this method, we ensure that data modifications are handled immutably, with each change creating a new version while retaining a verifiable history of all prior states. This has allowed us to implement precise tracking of data evolution over time and enabling robust querying capabilities. Content-based addressing has been instrumental in achieving both efficiency and reliability in our file system design.

Chapter 3

Components

3.1 Crawler

To facilitate efficient indexing and retrieval of `git` repository data, we developed a custom crawler designed to traverse the commit history and recursively process directory structures. The crawler leverages the `libgit2[? ]` library for interacting with `git` repositories and employs the `xxHash[? ]` library for rapid hashing of files. The primary goal of this crawler is to generate structured metadata and content addressable storage suitable for integration with virtual file systems, such as the one implemented later using `FUSE`.

3.1.1 Commit History Traversal

The crawler initiates its operation by traversing the commit history of the repository's primary branch (`main` or `master`). Utilizing `libgit2[? ]`, it iteratively accesses each commit, extracting the commit identifier and associated directory tree. For each commit, the crawler processes the directory structure recursively,

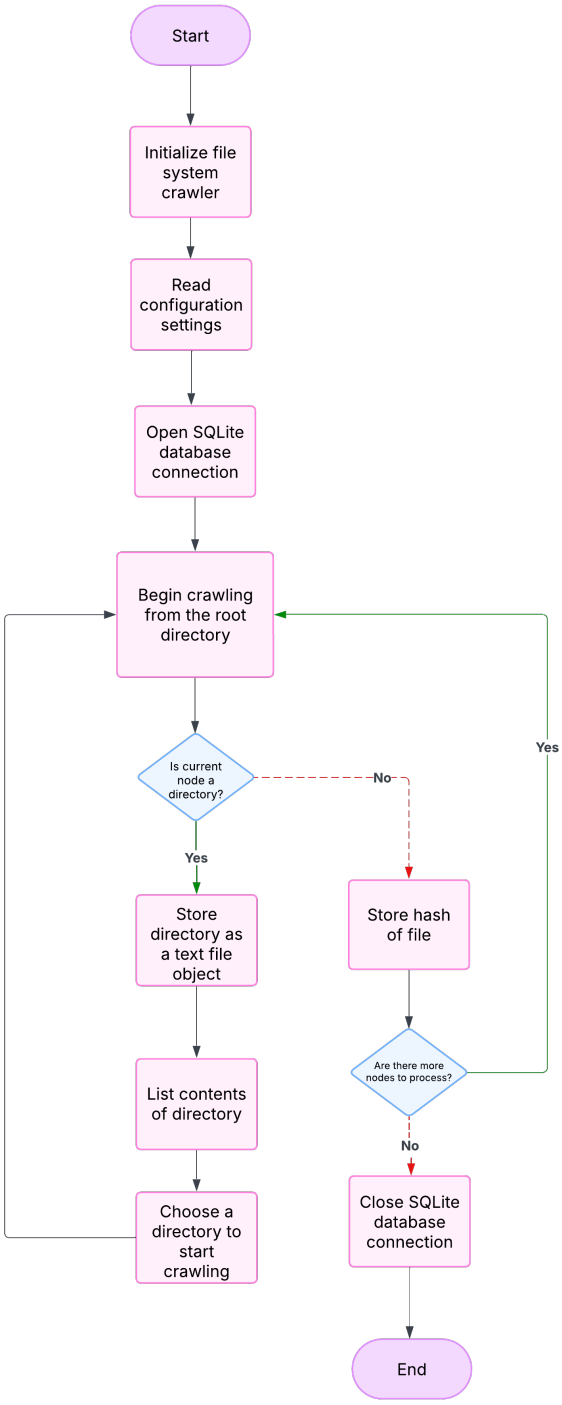

Figure 3.1: Flowchart for crawler

ensuring comprehensive coverage of the repository's historical states.

3.1.2 Recursive Directory and File Processing

During traversal, the crawler systematically processes each directory and file encountered. For files, it computes a unique hash using the `xxHash[? ]`, chosen specifically for its exceptional speed and efficiency in hashing large datasets. Each hashed file is then copied into a dedicated content directory, named according to its computed hash, thereby preventing duplication and facilitating rapid retrieval. For directories, the crawler generates a custom plain text `directory object`, which enumerates all immediate child files and subdirectories. Each entry within this directory object adheres to the following structured format:

```
filename|hash|0 (for files)
dirname|hash|1 (for directories)
```

Once generated, the directory object itself is hashed using `xxHash[? ]`, and the resulting hash serves as the unique identifier for the directory. This recursive hashing approach ensures that directory structures are accurately represented and efficiently retrievable.

3.1.3 Metadata Storage and Management

To facilitate efficient querying and retrieval, the crawler stores metadata (specially the commit ids) in an `SQLite` database (`fs.sqlite`). The database schema includes essential fields such as commit id, directory hashes, and relative paths from the repository root. This structured metadata storage enables fast lookups and supports retrieval operations.

3.1.4 Highlighted Features

The crawler's design offers these advantages:

- **Performance:** Leveraging `xxHash[? ]` ensures fast-hashing performance, particularly beneficial when processing large repositories or extensive commit histories.

- **De-duplication:** Storing files and directories based on their hashes inherently prevents duplication, optimizing storage utilization.

3.2 File System

The file system in `TimeFuseDB` is a critical component that integrates content-addressable storage (`CAS`) with temporal versioning, enabling efficient and transparent access to historical data. Built on top of `FUSE`, the system provides a virtual file system interface that allows users to interact with versioned content as if it were part of a traditional file system.

The `CAS` structure organizes files and directories into a sharded directory structure based on the first two characters of their hash. This approach optimizes file system performance by distributing files across multiple subdirectories, reducing the likelihood of performance bottlenecks caused by large numbers of files in a single directory.

`TimeFuseDB` introduces a temporal dimension to the file system, allowing users to access content as it existed at specific points in time. This is achieved through timestamp-based path resolution. For example, a path containing a timestamp pattern `(TIMESTAMP-<epoch_time>)` is interpreted as a request to retrieve the state

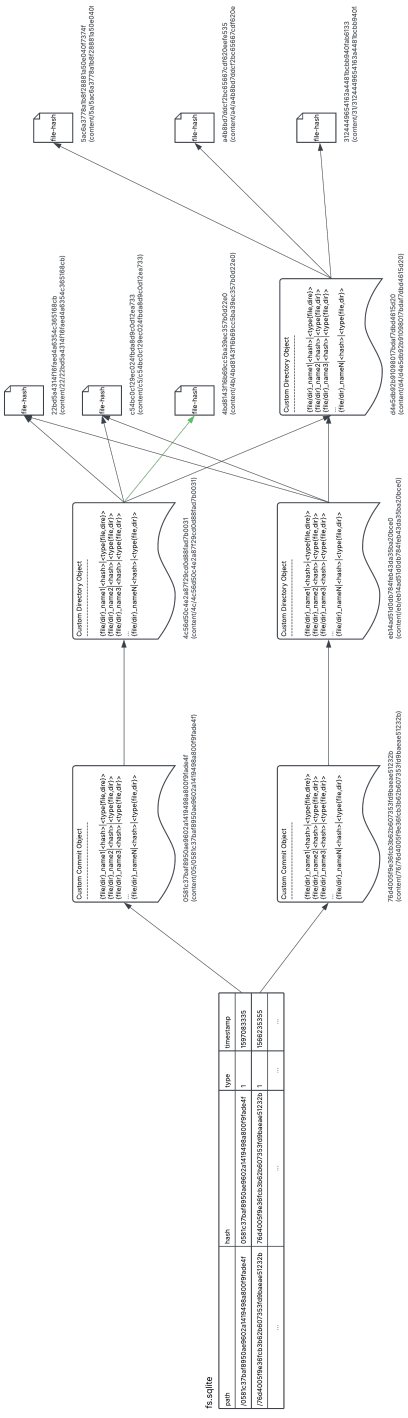

Figure 3.2: Sample internal structure of a `TimeFuseDB` file-system

of the file system at the specified time. The system performs a reverse lookup in the `SQLite` database to identify the corresponding content hash for the requested timestamp.

3.2.1 FUSE Implementation

`FUSE`, or File System in User Space, is a software interface that allows us to create and run fully functional file systems in user space rather than kernel space. This approach simplifies the process of file system development by enabling custom implementations without requiring kernel modifications. The architecture of `FUSE` consists of two primary components: a kernel module and a user-space application. The kernel module serves as an intermediary, forwarding system calls, such as `open`, `read`, and `write`, to the user-space file system. The user-space application, often using the `libfuse[? ]` library, implements the core file system logic and handles operations like file and directory management.

The `FUSE` operations dynamically resolve paths, retrieve metadata, and access content based on the requested version or timestamp. For instance, the `readdir` operation processes directory objects stored in the `CAS` to populate the directory listing. Directory objects are `plaintext` files that list the names, hashes, and types (file or directory) of their child elements. This design ensures that directory listings are **dynamically** generated based on the current or historical state of the file system.

To improve performance, `TimeFuseDB` employs an in-memory path cache that maps file paths to their corresponding content hashes and entry types (file or directory). This reduces the overhead of repeated database queries for frequently accessed paths.

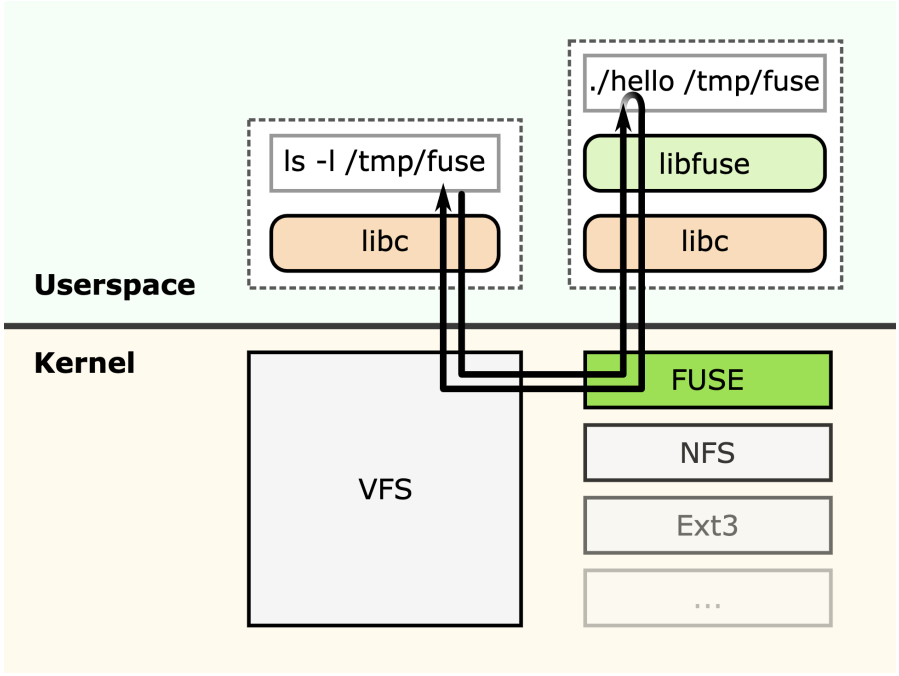

Figure 3.3: A flow-chart diagram showing how FUSE works [**?**]

Chapter 4

Use Cases

4.1 User-Friendly Temporal data Navigation

Users typically have well-developed skills for navigating traditional file systems using familiar tools and interfaces. In contrast, conventional version control systems like git introduce a significant learning curve due to their reliance on complex command sequences. TimeFuseDB addresses this difficulty by exposing temporal data directly through a familiar file system interface, allowing users to intuitively explore historical states of data without having to learn specialized versioning commands or data access patterns.

4.2 Management of Financial Configurations

In computational finance, model weights and configuration parameters are adjusted frequently—often daily or even multiple times per day—to adapt to rapidly changing market conditions. Regression testing of new models is crucial, as it enables practitioners to assess how these models would have performed under historical scenarios, sometimes requiring simulations that span multiple years of data. Accurately capturing the temporal evolution of these financial parameters is essential for high-fidelity simulations. TimeFuseDB addresses this need by automatically preserving every change in model weights and configuration settings over time. Its temporal file system interface lets users explore historical configurations as if browsing a traditional directory structure. This intuitive access eliminates the complexity of specialized database queries or version control commands while ensuring that every modification is reliably recorded for comprehensive backtesting and analysis.

4.3 Historical Securities Data Navigation

Financial securities evolve over time due to corporate actions such as mergers, stock splits, and other restructuring events that typically occur on trading day boundaries. When analyzing a security's long-term performance, it is critical to reference the correct version corresponding to the precise dates of interest. Traditionally, such dynamic historical data is managed with bi-temporal databases that record both effective dates and recording timestamps, but interfacing with these systems often demands specialized expertise and complex queries. While TimeFuseDB is currently designed as a uni-temporal system, its architecture offers

a promising foundation for future extension to support two time dimensions. By adapting TimeFuseDB to manage both effective and recorded time, users could intuitively navigate historical securities data through a familiar file system interface.

Chapter 5

Conclusions and Future Work

- **Enhancing Configurability and Usability**: Currently, configuration of `TimeFuseDB` demands modification of source code files (e.g., `.cpp` files). A significant future improvement is the development of a user-friendly and flexible configuration mechanism, such as dedicated configuration files or graphical user interfaces.

- **Improving Date and Time Format Handling**: Providing comprehensive support for various date and time formats can substantially improve the system's versatility and practical utility. Additional research could be conducted to integrate more standardized date-time representations, locale-specific formats, and enhanced temporal querying capabilities, thus improving user experience across diverse domains.

- **Maturing `TimeFuseDB` into a Production-Ready, Open-Source Product**: Transitioning from a research prototype into a polished, open-source solution is an important next step. Achieving this involves rigorous testing, detailed documentation, ease of installation, and robust error handling. Open-

sourcing the project will also encourage community engagement, external contributions, and real-world validation of its practical value.

- **Extending to Support Bi-temporal Data**: While the current design focuses on uni-temporal data management, a valuable future direction is extending `TimeFuseDB` to support multiple time dimensions (e.g., bi-temporal data). Adding a second temporal dimension (valid time in addition to system time) will enable more sophisticated historical tracking and auditing, particularly useful in fields like finance and regulatory compliance.

- **Interface Improvement**: A compelling future application of this file system-based approach is to innovate within the version control domain, potentially replacing traditional `git` front-ends. Given that `git`'s user experience is often criticized for complexity and steep learning curves, re-imagining version control through `TimeFuseDB`'s intuitive directory-based interface could markedly enhance usability, representing a transformative shift in how users interact with historical versions of data.

- **Serve `git`'s internal DB**: A promising direction for future work is to enhance `TimeFuseDB` by directly serving `git`'s native database with `FUSE` without duplicating data into a separate content-addressable store. This approach would significantly reduce storage and computational overhead.

Appendix A

Example Usage

This provides a detailed, step-by-step instructions for setting up and using the system. Note that `TimeFuseDB` currently operates on `Linux` or `macOS` platforms where `FUSE` is supported.

A.1 Repository and Code Setup

1. **Clone the Repository:**

   Begin by cloning the `TimeFuseDB` repository from `GitHub`:

   ```
   > $ git clone https://github.com/aniket-ray/TimeFuseDB.git
   ```

   Then, navigate into the repository directory:

   ```
   > $ cd TimeFuseDB
   ```

2. **Edit the Configuration File:**

   Open the file `common.cpp` with your preferred text editor. For example, using `vim`:

   ```
   > $ vim common.cpp
   ```

   Important Configuration Steps:

   - `ROOT_DIR`: Modify this path to point to your target git repository—the repository whose commit history you wish to analyze.
   - `CONTENT_PATH`: Change this to an empty directory where the system will store files using their computed hashes.

A.2 Building the Project

After configuring the source files, compile the project by following these steps:

1. Create a build directory:

   ```
   > $ mkdir build
   ```

2. Change into the build directory and generate build files using CMake:

   ```
   > $ cd build && cmake ..
   ```

3. Compile the project using `make`:

```
> $ make
```

This process generates two executables: `crawler` and `TimeFuseDB`.

A.3 Running the System

1. **Execute the Crawler:**

   Run the crawler to process your git repository. This step will:

   - Traverse the repository and extract the commit history.
   - Populate the `SQLite` database with metadata detailing the repository's evolution.
   - Compute the `XXH128` hash for each file and store the file in the directory specified in `CONTENT_PATH`.

   Execute the crawler with:

   ```
   > $ ./crawler
   ```

   Monitor the terminal output to follow the crawling progress.

2. **Mount the Virtual File System:**

   Once the crawling process is complete, use the `TimeFuseDB` executable to mount the virtual file system. The `-f` flag specifies the mount point on your host system:

```
> $ ./TimeFuseDB -f /path/to/your/mount/directory
```

FUSE dynamically generates the file system structure based on the `SQLite` database and the git commit history. The mounted file system recreates the state of the repository at various Unix epoch timestamps.

3. **Browsing the Commit History:**

   Within the mounted directory, we will find folders with the commit hashes. These directories represent commits of the repository. To inspect the state of the repository at a particular `EPOCH` (say `1618033988`) navigate to the root the mounted file-system and run:

   ```
   > $ cd TIMESTAMP-1618033988
   ```

   This navigation allows us to easily browse the historical state of your repository as maintained by the system.

Bibliography

[1] D. Mastromatteo, "Writing a fuse filesystem in python," The Python Corner, 02 2017. [Online]. Available: https://thepythoncorner.com/posts/2017-02-27-writing-a-fuse-filesystem-in-python/

[2] libgit2, "Github - libgit2/libgit2: A cross-platform, linkable library implementation of git that you can use in your application." GitHub, 05 2024. [Online]. Available: https://github.com/libgit2/libgit2

[3] "libfuse/libfuse," GitHub, 05 2021. [Online]. Available: https://github.com/libfuse/libfuse

[4] "File:fuse structure.svg - wikimedia commons," Wikimedia.org, 11 2007. [Online]. Available: https://commons.wikimedia.org/wiki/File:FUSE_structure.svg

[5] Y. Collet, "Github - cyan4973/xxhash: Extremely fast non-cryptographic hash algorithm," GitHub, 07 2023. [Online]. Available: https://github.com/Cyan4973/xxHash