DagShell: A Content-Addressable Virtual Filesystem with Multiple Interfaces

Anonymous Authors
Under Review

November 3, 2025

Abstract

We present DagShell, a virtual filesystem implementation that combines content-addressable storage with POSIX semantics. DagShell represents filesystem objects as immutable nodes in a directed acyclic graph (DAG), where each node is identified by its SHA256 hash. The system provides three distinct interfaces: a fluent Python API for programmatic access, a Scheme-based domain-specific language for scripting, and a full POSIX-compliant terminal emulator for interactive use. DagShell achieves complete immutability while maintaining familiar filesystem semantics, enabling automatic deduplication, complete history preservation, and deterministic filesystem states. The implementation demonstrates 99% test coverage and provides practical applications in sandboxing, version control experiments, and reproducible computational environments.

1 Introduction

Modern filesystems prioritize mutable state and in-place updates for performance reasons. However, several domains benefit from immutable, content-addressed storage: version control systems track file history through content hashes, package managers use cryptographic hashes to verify integrity, and distributed systems rely on content addressing for deduplication and verification. Despite these successes in specialized tools like Git [1], IPFS [2], and Nix [3], general-purpose filesystems remain fundamentally mutable.

DagShell explores an alternative design point: a fully immutable, content-addressable filesystem that maintains POSIX semantics. Unlike Git, which tracks file history, DagShell provides a complete virtual filesystem where every operation creates new immutable nodes. Unlike IPFS, which focuses on distributed content distribution, DagShell emphasizes familiar filesystem operations and multiple interface paradigms.

The key contributions of this work are:

- A content-addressable filesystem design that preserves POSIX semantics while ensuring complete immutability
- Three complementary interfaces (Python API, Scheme DSL, terminal emulator) demonstrating different interaction patterns
- Implementation techniques for efficient path resolution and state management in an immutable structure
- Evaluation of the system through comprehensive testing and practical use cases

2 Design

2.1 Core Architecture

DagShell's architecture separates content storage from path namespace management. The system maintains two primary data structures:

1. **Node store** (N : Hash $\rightarrow$ Node): A mapping from SHA256 hashes to immutable node objects

2. **Path index** (P : Path $\rightarrow$ Hash): A mapping from absolute paths to node hashes

This separation enables the same content to exist at multiple paths (deduplication) while providing familiar path-based access patterns.

2.2 Node Types

The system defines three node types, each immutable and content-addressed:

FileNode: Contains binary content and metadata (mode, uid, gid, mtime). The hash incorporates both content and metadata, ensuring that permission changes create new nodes:

```
@dataclass(frozen=True)
class FileNode(Node):
    content: bytes = b""
    mode: int = Mode.FILE_DEFAULT
    uid: int = 1000
    gid: int = 1000
    mtime: float = field(
        default_factory=time.time)
```

Listing 1: FileNode structure

DirNode: Contains a mapping from names to child node hashes. Adding or removing a child creates a new directory node with an updated children mapping:

```
@dataclass(frozen=True)
class DirNode(Node):
    children: Dict[str, str] = field
        (default_factory=dict)
    mode: int = Mode.DIR_DEFAULT
    # metadata fields...
```

Listing 2: DirNode structure

DeviceNode: Represents virtual devices (/dev/null, /dev/random, /dev/zero) with programmatic behavior rather than stored content:

```
def read(self, size: int = 1024) ->
   bytes:
    if self.device_type == 'null':
        return b''
    elif self.device_type == 'zero':
        return b'\x00' * size
    elif self.device_type == 'random
        ':
        return os.urandom(size)
```

Listing 3: DeviceNode read operation

2.3 Immutability and Copy-on-Write

All write operations follow a copy-on-write pattern. Writing to `/a/b/file.txt` requires creating:

1. A new FileNode with the updated content
2. A new DirNode for `/a/b` with updated children
3. A new DirNode for `/a` with updated children
4. An updated root DirNode

This propagation of changes up the directory tree maintains referential integrity while preserving all previous states. The path index is updated to point to the new node hashes, while old nodes remain in the node store until garbage collection.

2.4 Deduplication

Content addressing provides automatic deduplication at the node level. If two files have identical content and metadata, they produce the same hash and share storage:

```
fs.write("/file1.txt", "hello")
fs.write("/file2.txt", "hello")
# Both paths reference the same node
    hash
```

Listing 4: Deduplication example

This property extends to directories: two directories with identical children share the same node. Deduplication occurs transparently without explicit user intervention.

2.5 Soft Deletion and Garbage Collection

Deletion operates through two mechanisms:

Soft delete: Removes the path mapping but preserves nodes in the node store. This enables recovery and maintains referential integrity.

Garbage collection: The `purge()` operation performs mark-and-sweep garbage collection, removing nodes unreachable from the path index:

```
def purge(self) -> int:
    referenced = set()
    def mark(hash):
        if hash in referenced:
            return
        referenced.add(hash)
        node = self.nodes[hash]
        if node.is_dir():
            for child_hash in node.
                children.values():
                 mark(child_hash)

    for hash in self.paths.values():
        mark(hash)

    unreferenced = set(self.nodes.
        keys()) - referenced
    for hash in unreferenced:
        del self.nodes[hash]
    return len(unreferenced)
```

Listing 5: Garbage collection algorithm

3 Implementation

3.1 Core Filesystem Operations

The FileSystem class implements standard POSIX operations (open, read, write, mkdir, ls, rm, stat) while maintaining immutability. Each operation either queries existing nodes or creates new nodes through the copy-on-write mechanism.

Path resolution converts absolute paths to node hashes through the path index. The implementation handles special cases:

- Root directory (`/`) is immutable and always present
- Device files in `/dev` behave according to their type
- Soft-deleted paths return `None` during resolution

File handles provide a familiar interface while operating on immutable content. The FileHandle class buffers writes in memory and commits changes to new nodes on close:

```
with fs.open("/file.txt", "w") as f:
     f.write("content")
# On context exit, creates new
    FileNode
# and updates path index
```

Listing 6: File handle pattern

3.2 User and Permission Management

DagShell implements POSIX user and group semantics through special files in `/etc`:

- `/etc/passwd`: User database with uid, gid mappings
- `/etc/group`: Group database with membership information

The system performs permission checks by evaluating mode bits against user context (uid, gid set). Root user (uid 0) bypasses permission checks. This design enables realistic permission modeling while maintaining filesystem immutability.

3.3 Fluent Python API

The fluent API (dagshell_fluent.py) provides method chaining and Unix-style composition. The DagShell class maintains session state (current directory, environment variables, last result) and translates operations into FileSystem calls:

```
shell = DagShell()
shell.mkdir("/project")\
     .cd("/project")\
     .echo("# My Project").out("
       README.md")\
     .echo("main.py").out(".gitignore
       ")
```

Listing 7: Fluent API example

The CommandResult class enables piping by storing operation results and providing output redirection:

```
shell.ls("/etc")\
    .grep("pass.*")\
    .sort()\
    .out("/tmp/sorted.txt")
```

Listing 8: Pipeline composition

Text processing commands (grep, sed, sort, uniq, wc, head, tail) operate on CommandResult objects, enabling Unix-style data flow without shell parsing.

3.4 Scheme Interpreter Integration

The Scheme interpreter (scheme_interpreter.py) provides a functional programming interface to filesystem operations. The implementation includes:

- Complete Scheme evaluator with lexical scoping
- Built-in filesystem procedures mapped to Python operations
- List processing and higher-order functions for file manipulation

```
(begin
  (mkdir "/project")
  (write-file "/project/README.md"
              "# Project")
  (map (lambda (f)
         (write-file
           (string-append "/backup/"
               f)
           (read-file f)))
       (find "/src" "*.py")))
```

Listing 9: Scheme filesystem example

The Scheme environment includes filesystem operations (mkdir, write-file, read-file), text processing (grep, head, tail), and functional primitives (map, filter, reduce). This enables scripting complex filesystem operations in a functional style.

3.5 Terminal Emulator

The terminal emulator (terminal.py) provides a POSIX-compliant shell interface. The implementation separates concerns:

1. **CommandParser**: Parses shell syntax (pipes, redirects, operators)
2. **CommandExecutor**: Translates parsed commands to fluent API calls
3. **TerminalSession**: Manages REPL loop and session state

The parser handles standard shell features:

- Pipes: `ls | grep txt | sort`
- Redirects: `echo "data" > file.txt`
- Logical operators: `mkdir dir && cd dir`
- Sequential execution: `cmd1 ; cmd2`

The executor maps command names to shell methods, handles flag translation, and applies redirections to CommandResult objects. This architecture enables shell-like interaction while leveraging the fluent API for implementation.

3.6 Slash Commands

The terminal provides special slash commands for meta-operations:

Host Integration:

- `/import <host> <virt>`: Import files from host filesystem with safety restrictions
- `/export <virt> <host>`: Export to host filesystem

State Management:

- `/save <file>`: Serialize to JSON
- `/load <file>`: Deserialize from JSON
- `/snapshot <name>`: Create timestamped snapshot

DAG Inspection:

- `/status`: Show filesystem statistics
- `/dag`: Visualize DAG structure
- `/nodes`: List content-addressed nodes

- `/info <hash>`: Show node details

These commands operate outside the POSIX abstraction, providing direct access to the underlying DAG structure and host system integration.

4 Features and Capabilities

4.1 POSIX Compliance

DagShell implements a substantial subset of POSIX filesystem operations:

- File operations: open, read, write, close, stat
- Directory operations: mkdir, ls, rmdir
- Path operations: cd, pwd, absolute/relative path resolution
- Permission checking: Unix-style mode bits and ownership
- Special files: device nodes with appropriate semantics

The implementation prioritizes correctness over performance, ensuring that operations behave consistently with POSIX specifications where applicable.

4.2 Persistence and Serialization

The filesystem serializes completely to JSON format. The serialization includes:

- All nodes with their content (base64-encoded) and metadata
- Path-to-hash mappings
- Deleted path set for soft deletes

JSON serialization enables easy inspection, version control of filesystem states, and portability across systems. The deserialization process reconstructs node objects and indices, restoring complete filesystem state.

4.3 Import and Export

DagShell interfaces with the host filesystem through controlled import/export:

Import reads files from the host filesystem and creates corresponding nodes in the virtual filesystem. The operation:

- Preserves file modes and permissions
- Recursively imports directory trees
- Restricts access to safe directories
- Applies specified uid/gid to imported files

Export writes virtual filesystem content to the host. The operation:

- Creates directories and files on the host
- Preserves permissions when possible
- Maps virtual uid/gid to host uid/gid
- Reports the number of exported items

Safety mechanisms prevent unrestricted host access. Import and export operations validate paths against configured safe directories, preventing directory traversal attacks.

4.4 History and Versioning

Although DagShell doesn't implement explicit versioning, the immutable DAG structure preserves complete history. Combined with JSON serialization, users can:

- Save filesystem snapshots at any point
- Compare snapshots to identify changes
- Restore previous states by loading snapshots
- Track node-level changes through hash comparison

This design enables version control experimentation and rollback capabilities without implementing a full version control system.

5 Use Cases

5.1 Testing and Sandboxing

DagShell provides isolated filesystem environments for testing. Applications can:

- Test filesystem interactions without touching the host
- Create complex directory structures programmatically
- Reset to known states between test runs
- Verify filesystem operations through inspection

The Python API integrates naturally with pytest and other testing frameworks. The complete test suite demonstrates this approach, achieving 99% code coverage through isolated filesystem tests.

5.2 Educational Applications

DagShell serves as a teaching tool for filesystem concepts:

- Students explore content-addressable storage without distributed systems complexity
- The DAG structure makes relationships between files and directories explicit
- Immutability demonstrates functional programming principles in systems context
- Multiple interfaces show different abstraction levels over the same core

The Scheme interface particularly suits functional programming education, demonstrating how traditional imperative operations (filesystem modifications) can be expressed functionally.

5.3 Data Pipeline Experiments

The fluent API enables data processing pipelines with rollback:

```
shell = DagShell()
shell.save("before.json")

# Process data
shell.cat("/data/input.csv")\
    .grep("active")\
    .sort()\
    .out("/data/filtered.csv")

# Can rollback if needed
shell.load("before.json")
```

Listing 10: Data pipeline with snapshots

Researchers can experiment with data transformations, saving intermediate states and comparing results across different processing strategies.

5.4 Reproducible Environments

DagShell enables reproducible computational environments through:

- Complete filesystem state serialization
- Deterministic hashing ensures identical content produces identical hashes
- JSON format integrates with version control
- Import/export enables moving between virtual and real filesystems

A research workflow might maintain project data and scripts in DagShell, serialize the entire state to version control, and restore the exact environment on different machines.

6 Evaluation

6.1 Test Coverage

The implementation includes comprehensive test suites covering:

- Core filesystem operations (test_dagshell.py, test_core_filesystem_comprehensive.py)

- Fluent API and pipelines (test_fluent.py)
- Terminal emulation (test_terminal.py, test_enhanced_terminal.py)
- Scheme interpreter (test_scheme_interpreter.py, test_scheme_integration_comprehensive.py)
- Edge cases and error handling (test_edge_cases_comprehensive.py)
- Persistence and import/export (test_persistence_comprehensive.py, test_import_export_comprehensive.py)

Test coverage analysis reports 99% code coverage, indicating thorough testing of core functionality and edge cases. The test suite includes over 300 individual test cases exercising different aspects of the system.

6.2 Performance Characteristics

DagShell prioritizes correctness and simplicity over performance. Key performance characteristics:

Node creation: $O(1)$ time complexity for adding nodes to the store. Hashing dominates the cost, with SHA256 computation proportional to content size.

Path resolution: $O(k)$ where k is the path depth, requiring dictionary lookups through the path index.

Write operations: $O(d)$ where d is directory depth, due to copy-on-write propagation up the directory tree.

Garbage collection: $O(n)$ where n is the total number of nodes, using mark-and-sweep traversal.

Serialization: $O(n)$ for serializing all nodes to JSON, with base64 encoding overhead for binary content.

The in-memory design eliminates I/O overhead but limits filesystem size to available memory. For the target use cases (testing, education, experimentation), this tradeoff proves acceptable.

6.3 Memory Usage

Memory consumption scales linearly with content size and number of unique nodes. Deduplication reduces memory usage when multiple paths reference identical content. The Python object overhead (approximately 56 bytes per object) adds constant overhead per node.

For typical test scenarios (hundreds of files, megabytes of content), memory usage remains well within modern system constraints. Large-scale usage would require external storage and selective loading.

7 Related Work

7.1 Git

Git [1] pioneered practical content-addressed storage for version control. Git represents commits, trees, and blobs as content-addressed objects in a similar DAG structure. Key differences:

- Git tracks history explicitly through commit objects; DagShell provides snapshots without built-in history
- Git optimizes for large repositories through pack files; DagShell maintains individual nodes
- Git focuses on source code; DagShell provides a general filesystem interface

DagShell applies Git's content-addressing principles to a full POSIX-like filesystem rather than specialized version control.

7.2 IPFS

IPFS [2] implements distributed content-addressed storage with a focus on peer-to-peer data sharing. IPFS and DagShell share content-addressing principles but differ in scope:

- IPFS emphasizes distribution and discovery; DagShell focuses on single-system filesystem semantics

- IPFS uses Merkle DAGs for verification; DagShell uses SHA256 hashing for deduplication
- IPFS integrates with networking protocols; DagShell provides programming interfaces

DagShell can be viewed as exploring content-addressing without the complexity of distributed systems.

7.3 Nix

The Nix [3] package manager uses content-addressed storage for reproducible package management. Nix stores packages in `/nix/store` with paths derived from content hashes. Similarities include:

- Both use content addressing for determinism and deduplication
- Both enable rollback through immutability
- Both provide isolation from the system

However, Nix focuses on package management with derivations and dependency tracking, while DagShell provides a general filesystem interface.

7.4 Plan 9

Plan 9's [4] ”everything is a file” philosophy influenced DagShell's device node design. Plan 9 represents processes, networks, and system services as files in synthetic filesystems. DagShell adopts this approach for virtual devices while focusing on content-addressability rather than distribution.

7.5 Union Filesystems

Union filesystems like UnionFS and OverlayFS provide copy-on-write semantics over existing filesystems. These systems optimize write operations through layering. DagShell differs by:

- Maintaining all nodes explicitly rather than differencing layers
- Using content hashes for identification rather than layer ordering
- Providing multiple programming interfaces rather than FUSE mounting

8 Future Work

8.1 Garbage Collection Strategies

The current mark-and-sweep garbage collection runs synchronously. Future work could explore:

- Incremental garbage collection for large filesystems
- Reference counting to avoid full traversal
- Generational collection based on node age
- User-controlled retention policies for history

8.2 Compression

Content compression could reduce memory usage and serialization size. Strategies include:

- Transparent compression of file node content
- Delta compression for similar files
- Dictionary-based compression for common patterns

The tradeoff between compression overhead and memory savings warrants investigation.

8.3 Distributed Operation

While DagShell currently operates on a single system, the content-addressed design enables distribution:

- Node sharing across systems through hash-based lookup
- Merkle tree verification for integrity
- Synchronization protocols for state replication

- Conflict resolution for concurrent modifications

This direction would position DagShell closer to IPFS while maintaining its filesystem abstraction.

8.4 Performance Optimization

Several optimizations could improve performance:

- Path index caching for frequently accessed paths
- Lazy loading for large directory trees
- Binary serialization format replacing JSON
- Persistent storage backend with memory-mapped files

These optimizations would enable larger-scale usage while maintaining the design principles.

8.5 Enhanced Versioning

Explicit version control features could leverage the existing DAG:

- Commit objects linking snapshots with metadata
- Branch and merge operations
- Diff computation between filesystem states
- History visualization tools

This work would create a Git-like system with full filesystem semantics rather than just file tracking.

9 Conclusion

DagShell demonstrates that content-addressable storage principles can extend to general-purpose filesystem semantics. The system achieves complete immutability while maintaining familiar POSIX operations, automatic deduplication, and complete state preservation. Three distinct interfaces—fluent Python API, Scheme DSL, and POSIX terminal—show that content-addressing doesn't constrain interaction patterns.

The implementation proves practical for testing, education, and experimentation use cases. High test coverage and comprehensive functionality validate the design. While performance limitations constrain large-scale deployment, the core concepts apply to broader contexts.

Content-addressing offers benefits beyond distributed systems: automatic deduplication, hash-based verification, and implicit versioning enhance reliability and reproducibility. DagShell explores these benefits in a filesystem context, suggesting directions for future systems that combine traditional filesystem semantics with content-addressed storage.

The system's availability as open source enables further experimentation and extension. We hope DagShell inspires exploration of alternative filesystem designs that prioritize immutability, content-addressing, and reproducibility alongside traditional concerns like performance and scalability.

Acknowledgments

This work builds on ideas from Git, IPFS, Nix, and other content-addressed systems. The implementation benefited from Python's excellent support for immutable data structures and testing frameworks.

References

[1] Linus Torvalds and Junio C Hamano. *Git: Fast Version Control System.* `https://git-scm.com/`, 2005.

[2] Juan Benet. *IPFS - Content Addressed, Versioned, P2P File System.* arXiv:1407.3561, 2014.

[3] Eelco Dolstra. *The Purely Functional Software Deployment Model.* PhD thesis, Utrecht University, 2006.

[4] Rob Pike, Dave Presotto, Sean Dorward, Bob Flandrena, Ken Thompson, Howard Trickey, and Phil Winterbottom. *Plan 9 from Bell Labs.* Computing Systems, 8(3):221-254, 1995.

[5] Ralph C. Merkle. *A Digital Signature Based on a Conventional Encryption Function.* CRYPTO, 1987.

[6] Sean Quinlan and Sean Dorward. *Venti: A New Approach to Archival Data Storage.* USENIX FAST, 2002.

[7] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. *Secure Data Deduplication.* ACM StorageSS, 2008.

[8] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad N. Zubair. *Versatility and Unix Semantics in Namespace Unification.* USENIX ATC, 2004.

[9] Jeff Bonwick and Bill Moore. *ZFS: The Last Word in File Systems.* Sun Microsystems, 2007.