Git Virtual File System (GVFS)

Saeed Noursalehi sanoursa@microsoft.com

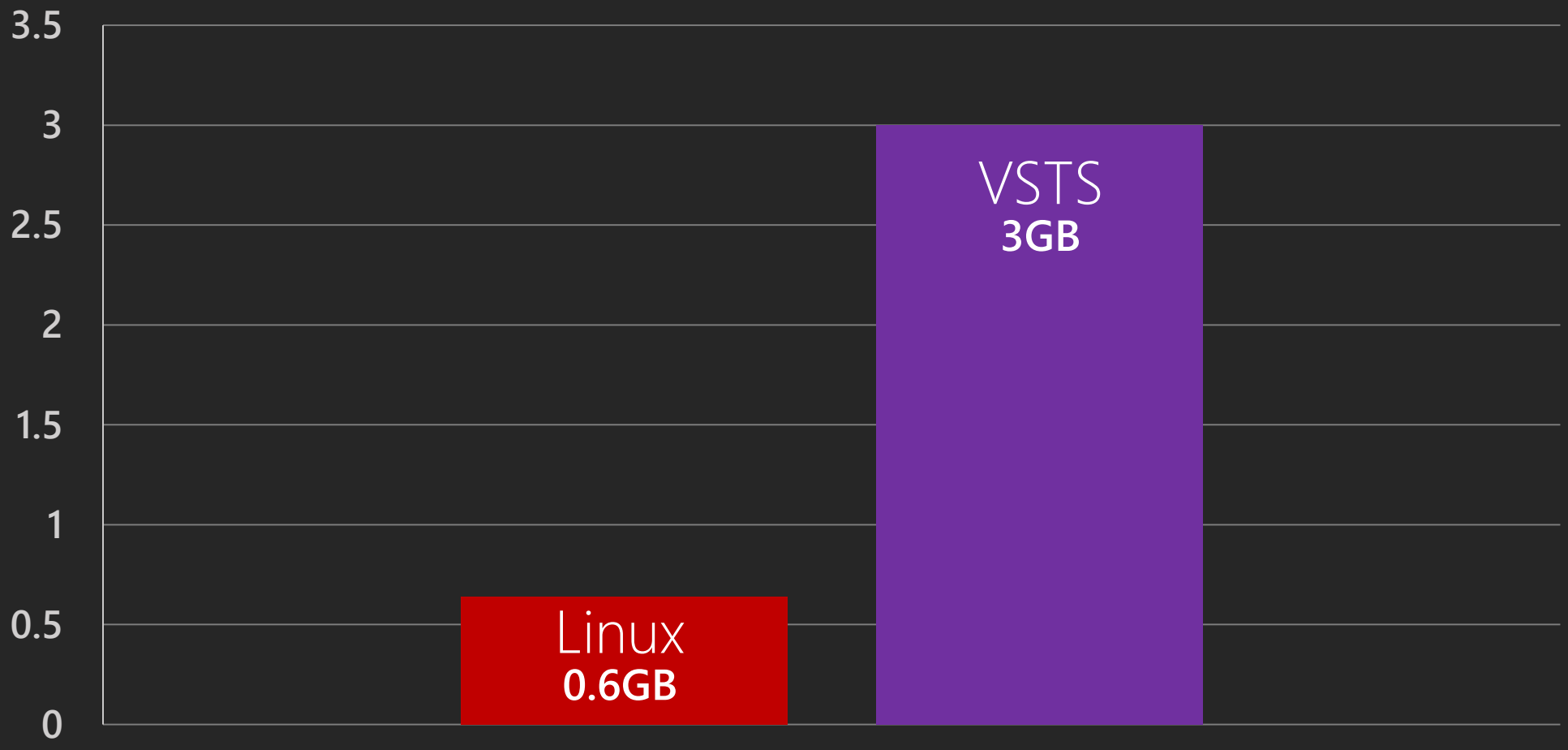
Visual Studio Team Services @ Microsoft

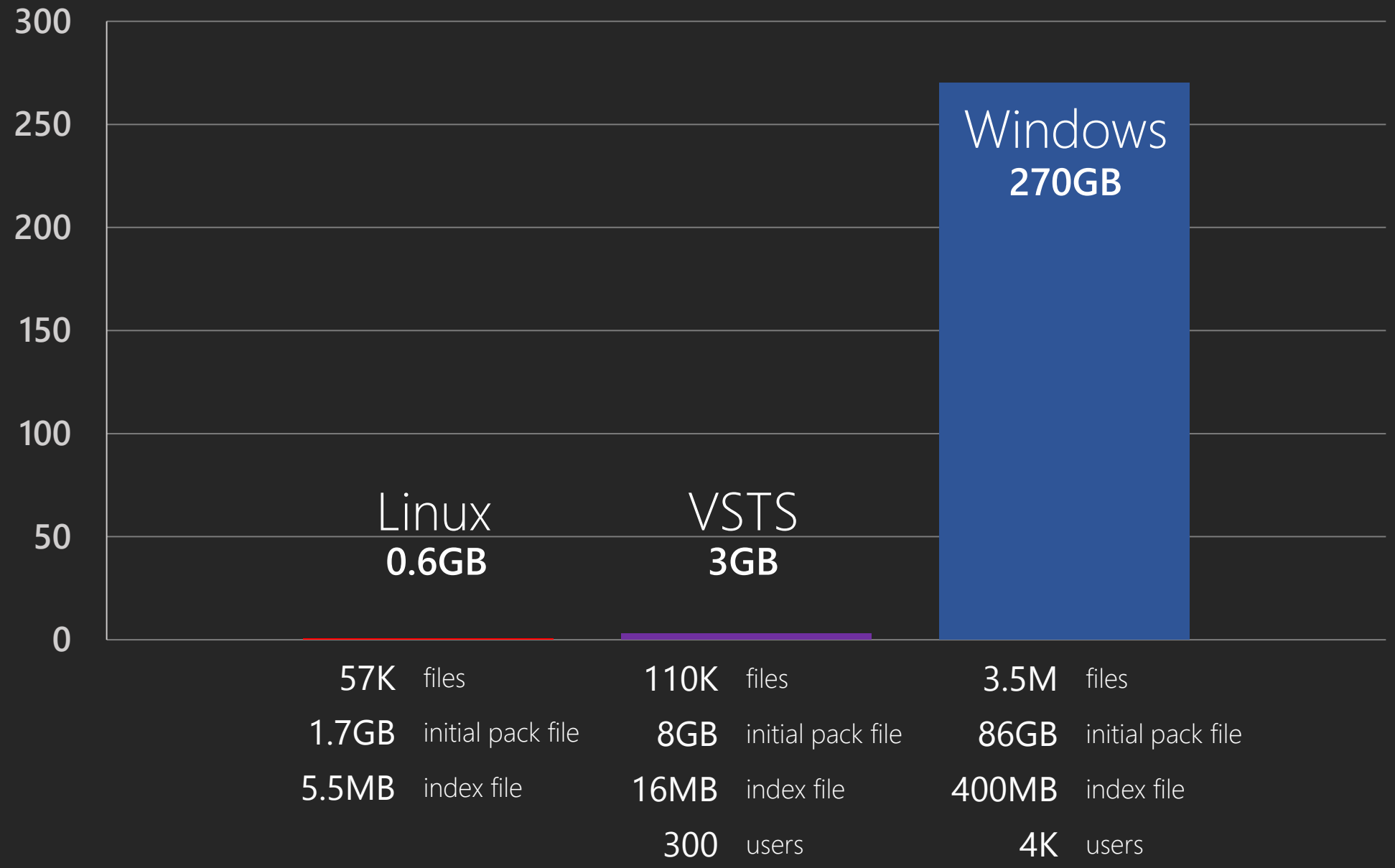
Agenda

- The Need for File System Virtualization
- Git background
- NTFS background
- GVFS Architecture

The Need for FS Virtualization

Two examples of large Git repositories




300
250
200
150
100
50
0
Linux
0.6GB
VSTS
3GB
Windows
270GB
57K files
1.7GB initial pack file
5.5MB index file
110K files
8GB initial pack file
16MB index file
300 users
3.5M files
86GB initial pack file
400MB index file
4K users

Perf of Git commands on Windows repo

12hrs	**clone**
3hrs	**checkout**
8mins	**status**
30mins	**commit**

Git background

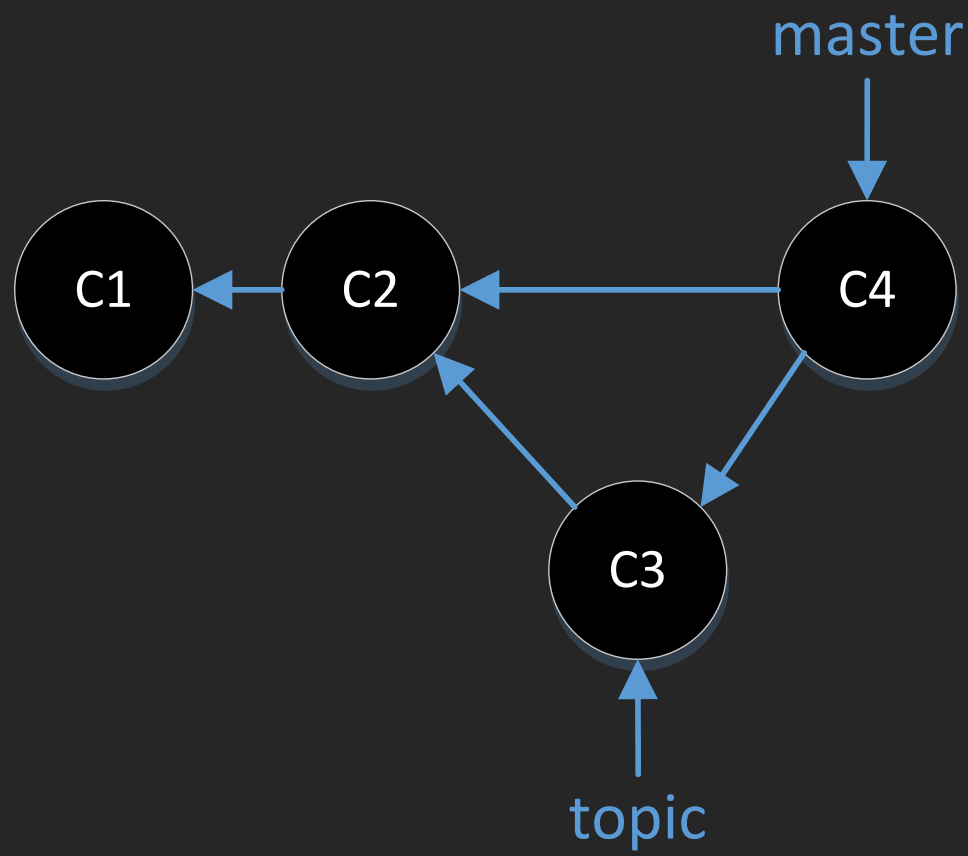
Git is a Distributed Version Control System

Basic concepts

- https://git-scm.com/book/en/v2

- Some very basic concepts (focusing on storage):
  - History is a Directed Acyclic Graph of commits
  - Branches are pointers to those commits
  - Every object is content addressable, with a sha1 hash

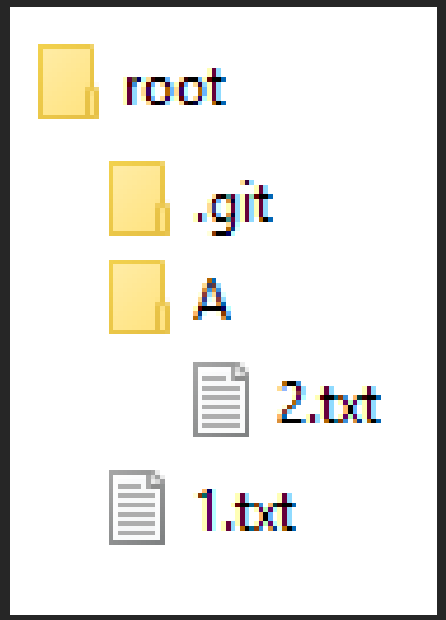
Commits form a DAG

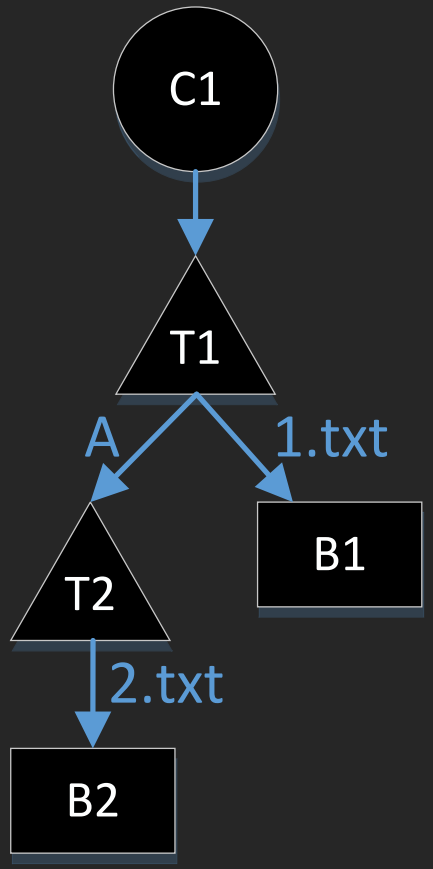


Branches are pointers to commits in the DAG

Trees and Blobs

Every commit points to one tree, representing the root of the working directory
Trees point at other trees and blobs





Git objects

- Commit
  - Represents the entire state of the working directory at a moment in time
  - Points at a single tree object representing the root folder
- Tree
  - Roughly equivalent to a folder
  - Points to other trees and blobs
- Blob
  - Roughly equivalent to a file

Some basic Git commands

- clone
  - Copy all history from server to client, then checkout most recent commit
  - Normally includes downloading all blobs, including historical ones
- checkout
  - Place all the files from a given commit into the working directory
- status
  - Scan through all files in working directory to figure out which are dirty
- commit
  - Take the current state of all files in the working directory and construct a new commit object from it

The Need for FS Virtualization

Problem statement …

- Git suffers from:

- Too many files
  - Status and checkout have to scan every single file in the repo
  - Approaching 4M files in the Windows repo
- Too much content
  - Clone and fetch have to download too much content
  - ~300GB in the Windows repo

… we tried lots of different solutions …

Go to https://www.visualstudio.com/learn/gvfs-design-history/ to read up on all the things we tried before we built GVFS

… solution

- Virtualize the file system
  - Too many files → Hide unmodified files from Git
  - Too much content → Only download the contents that are accessed

Benefits of virtualizing

- Support very large repos
  - No arbitrary boundaries in the codebase
  - Splitting the repo into many smaller repos fixes some scale problems, but creates new ones as well
- We can use stock Git
  - Using standard tools is far better than building new ones
- All existing tools (build, IDE) are unaware of the change

NTFS background

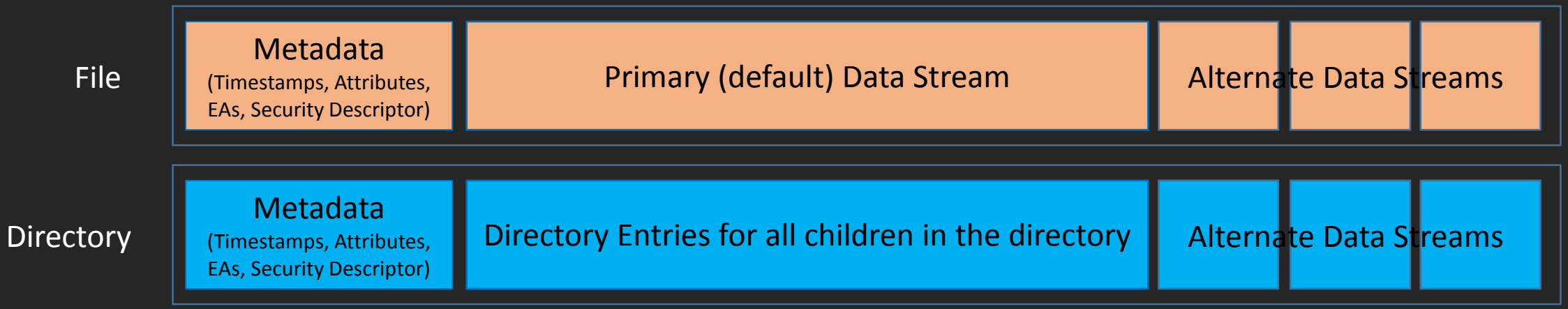
NTFS is the primary file system on Windows

Windows File System Drivers

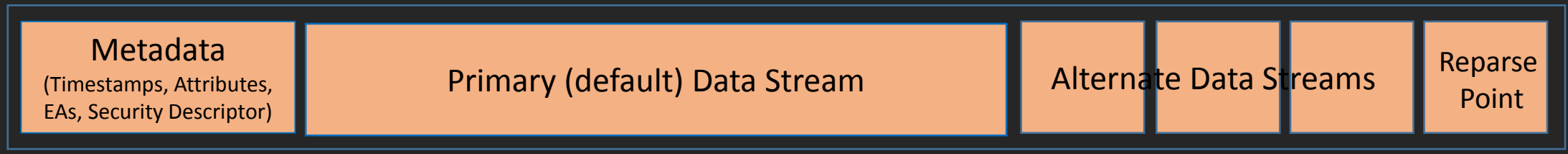
- A file system is implemented by a file system driver, e.g. ntfs.sys
- A file system driver can have zero or more file system *filter* drivers stacked on top of it

- Filter drivers intercept calls intended for the FS driver
  - The can pass through, redirect, modify, or reject those calls
  - Transparent to the user of the file system

File system objects



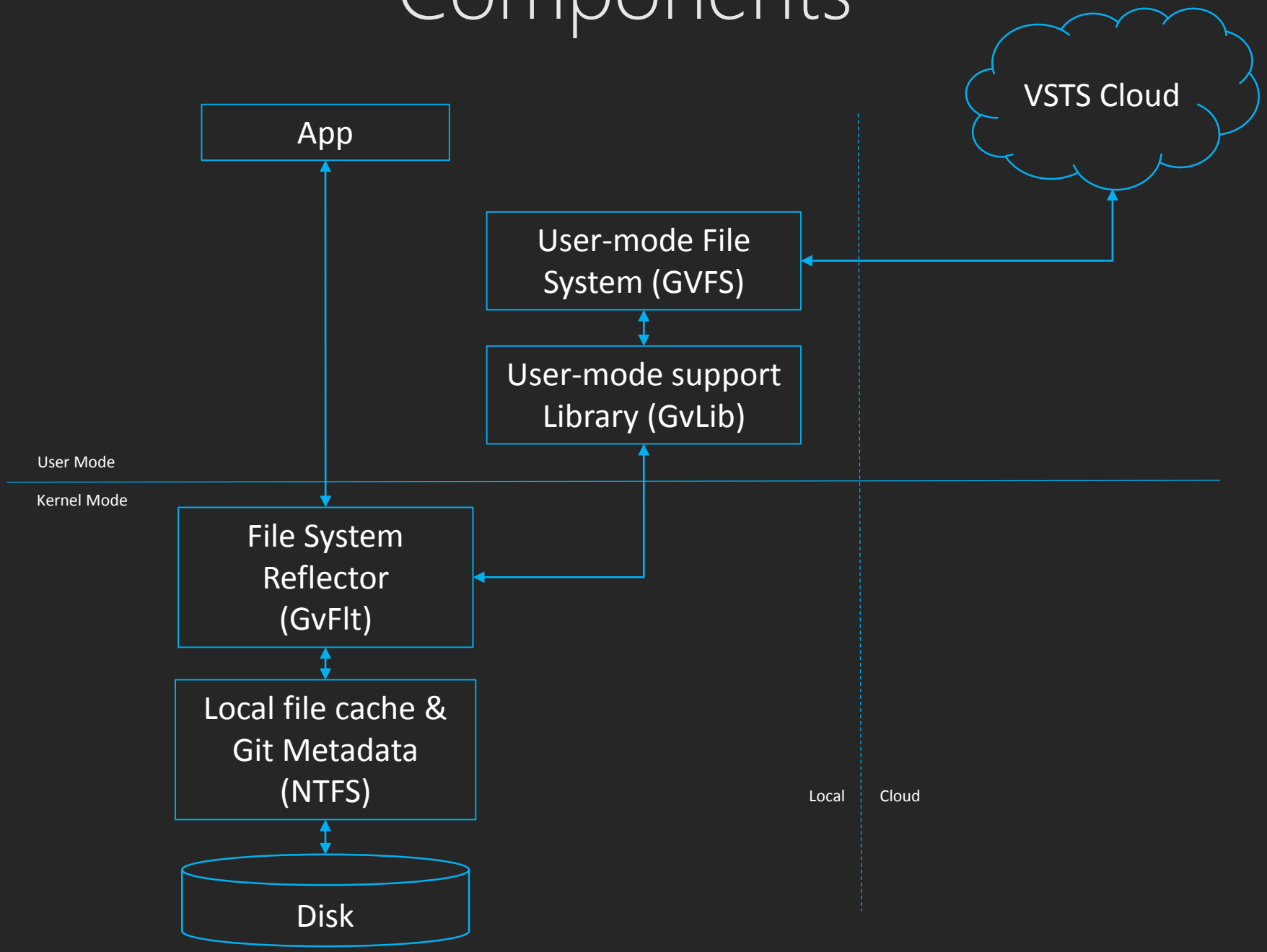
Reparse points



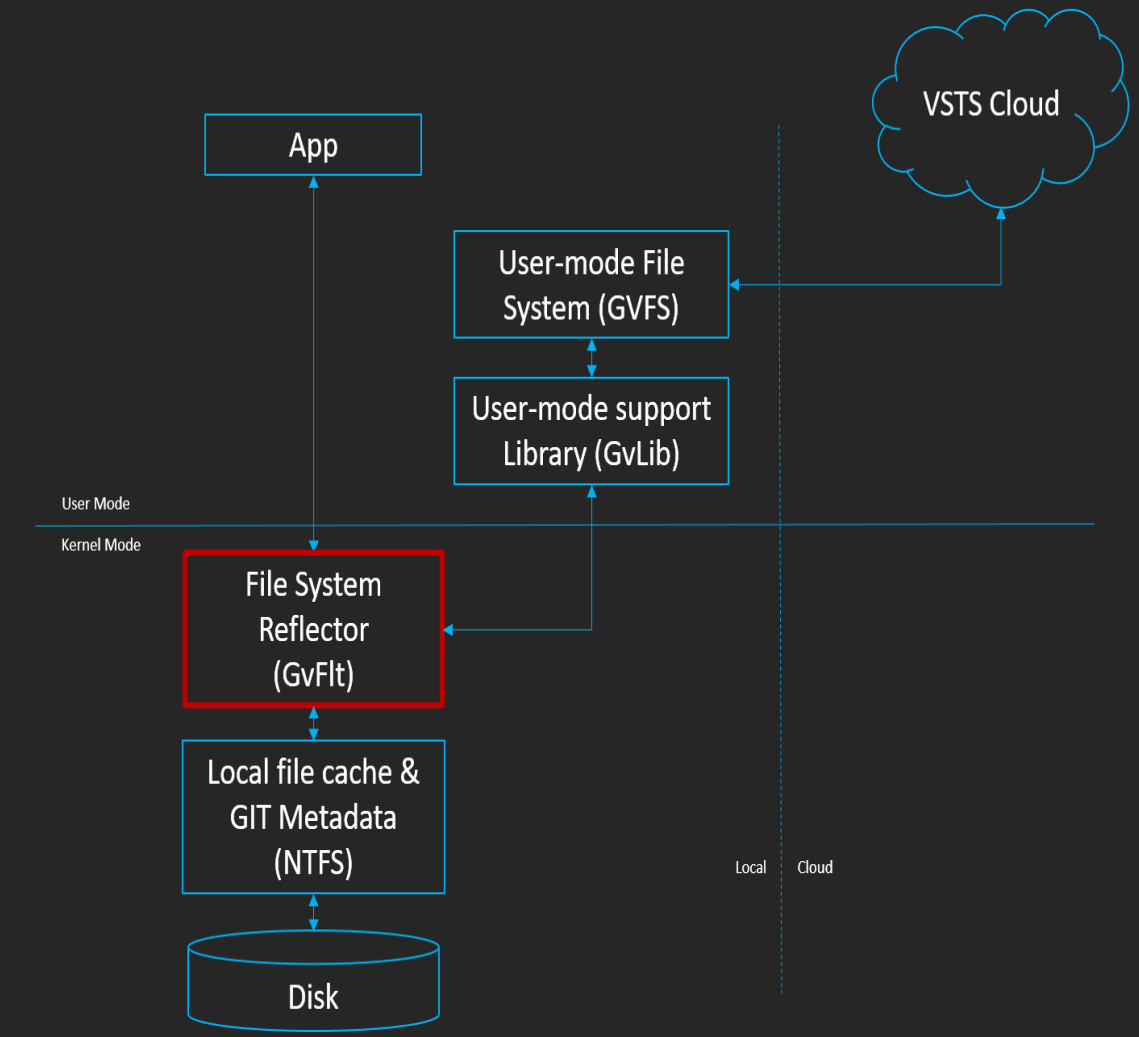
- Extra metadata on a file object
- Informs NTFS that a filter driver owns this file

GVFS Architecture

Components

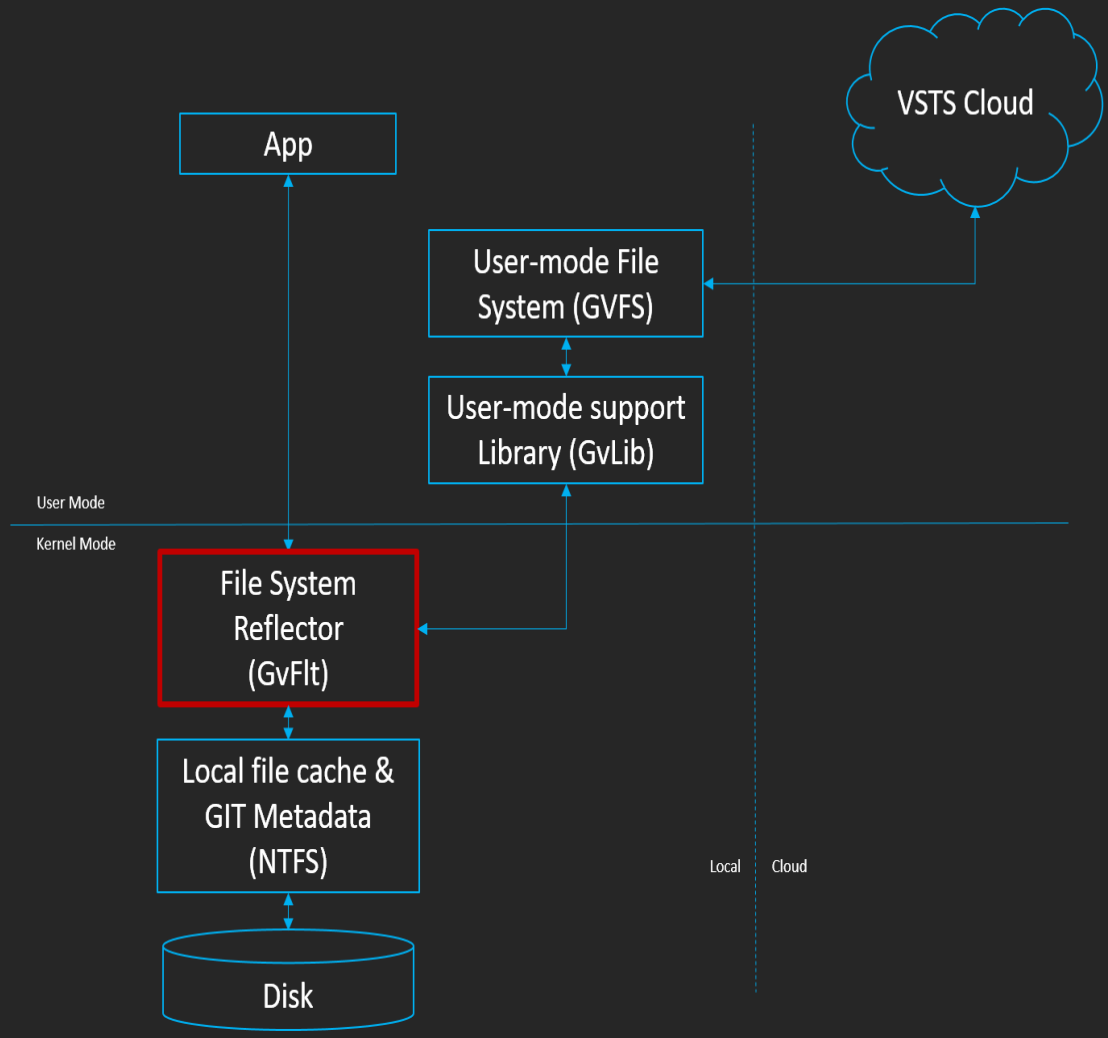

VSTS Cloud
App
User-mode File System (GVFS)
User-mode support Library (GvLib)
User Mode
Kernel Mode
File System Reflector (GvFlt)
Local file cache & Git Metadata (NTFS)
Disk
Local
Cloud

The File System Reflector (GvFlt)



Key Concepts and Functionality

- Kernel-mode file system filter driver
- Optimized callbacks from kernel-mode reflector to user-mode file system
- Maintains a local file cache in NTFS
- Unions local file cache with view projected by the user-mode file system



State of Items in the Local File Cache

Virtual
- Does not exist on local disk
- Projected during enumerations of parent directory

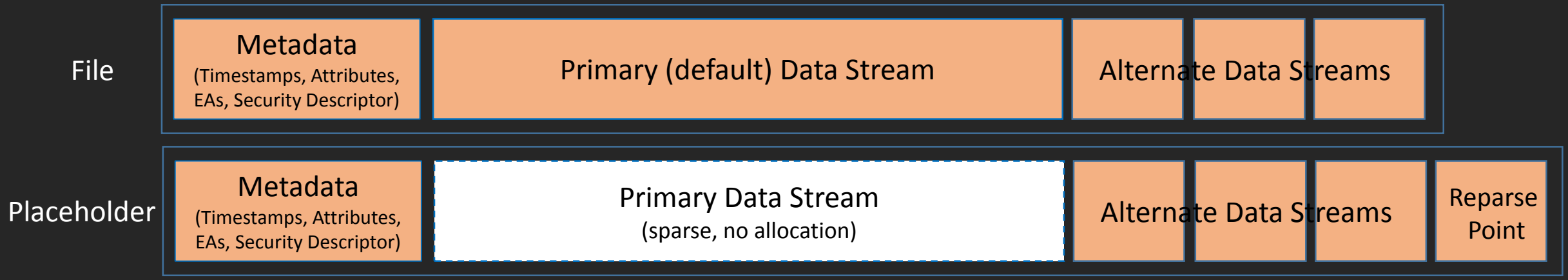
Placeholder
- No primary data content on disk
- Metadata on local disk as cache

Hydrated Placeholder
- Metadata and primary data content on disk as a cache

Full file
- Primary data content modified (no longer a cache)

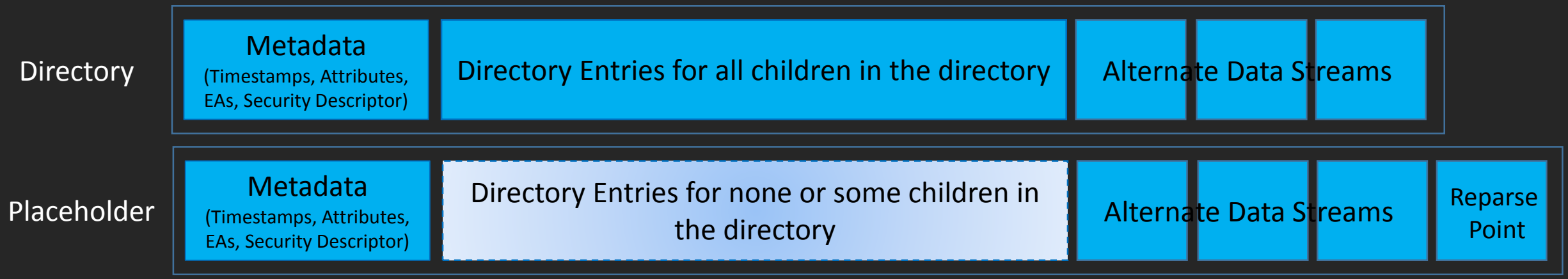
Tombstone
- Item deleted (no longer a cache)

Placeholder Files

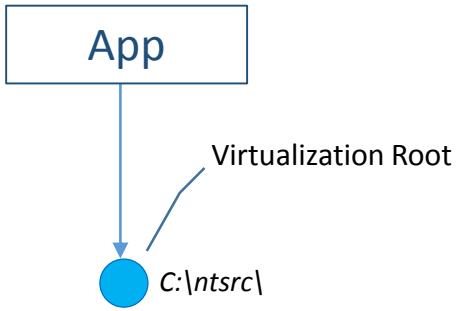


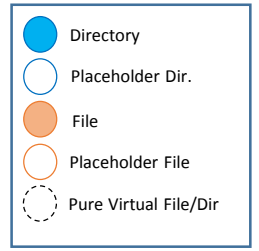
- Contains
  - Reparse point (but not a name redirector like symlink or junction)
  - Timestamps, attributes, alternate data streams, Eas
  - Primary data stream is sparse with correct EOF and VDL
- Share access checks, oplocks, byte range locks are all established on the Placeholder
- Hydrated on:
  - first read of primary data stream
  - open for write
  - Specific FS operations on a file (e.g. oplock, byte-range lock)
- All IO on hydrated placeholder is passthrough to the file system for native FS performance

Placeholder Directories

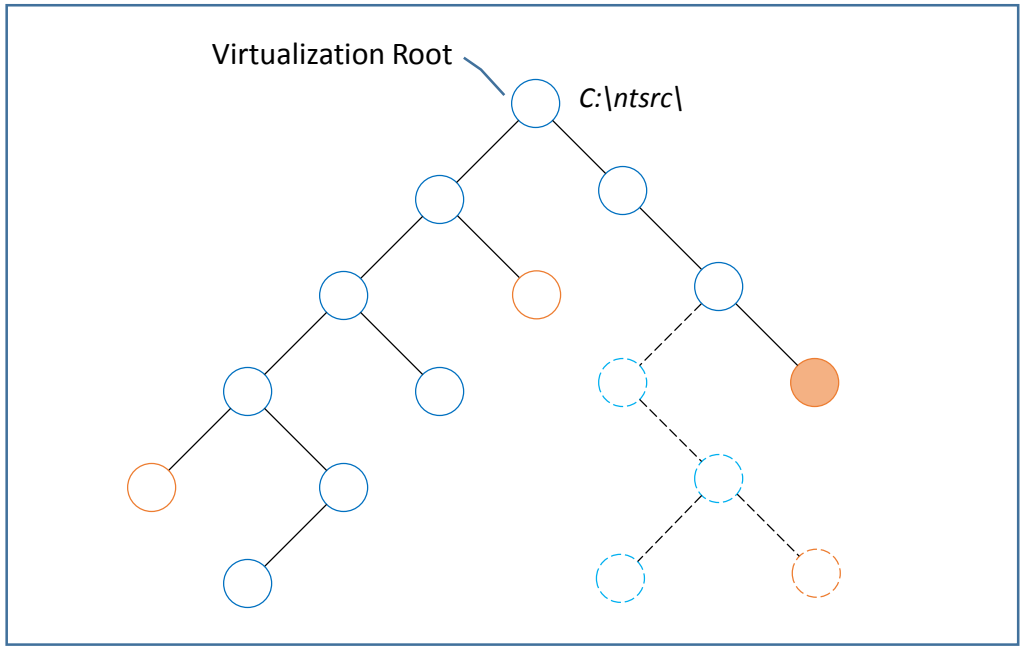


- Contains
  - Reparse point (but not a name redirector like symlink or junction)
  - Timestamps, attributes, alternate data streams, Eas
  - Contains no directory entries or directory entries for some children
- Leverages new feature in NTFS
  - Directory with reparse points can now have children
- Share access checks, oplocks are established on the Placeholder
- Enumerations are reflected to / merged with the directory view from user-mode file system
- Tombstones record deleted entries (deletes and renames of child items)

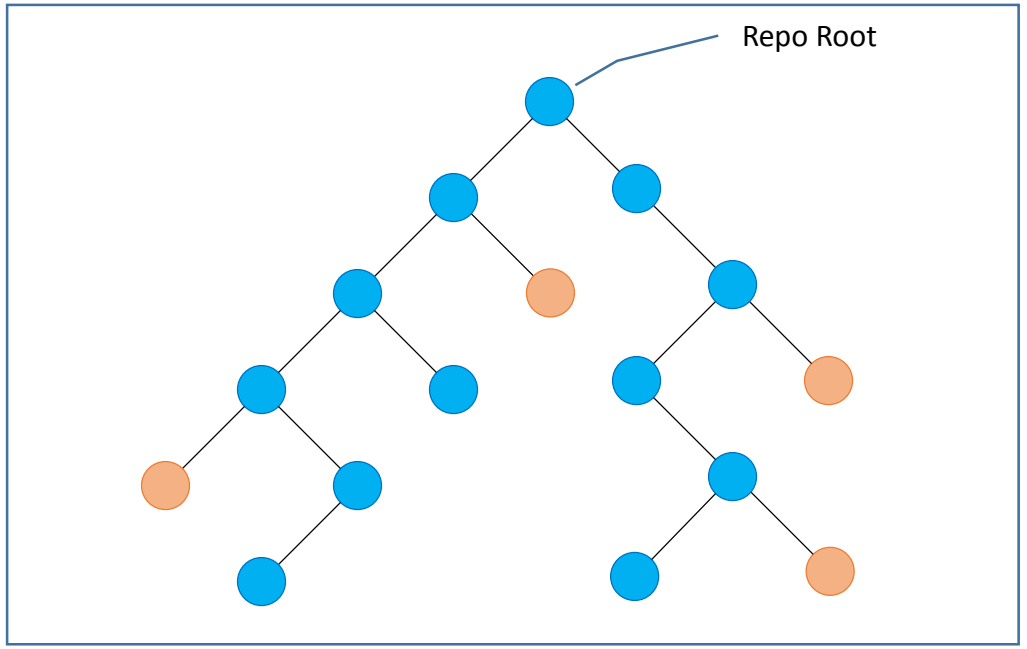




FS Union / Overlay



On-disk Local File
Cache in NTFS



Virtual View Projected
By User-Mode FS

1. Virtualization Callbacks

Directory Enumeration
- Start
- Get Directory Entries
- Stop

Placeholder creation
- Get Placeholder Information

File Hydration
- Get File Stream

2. Notification Callbacks

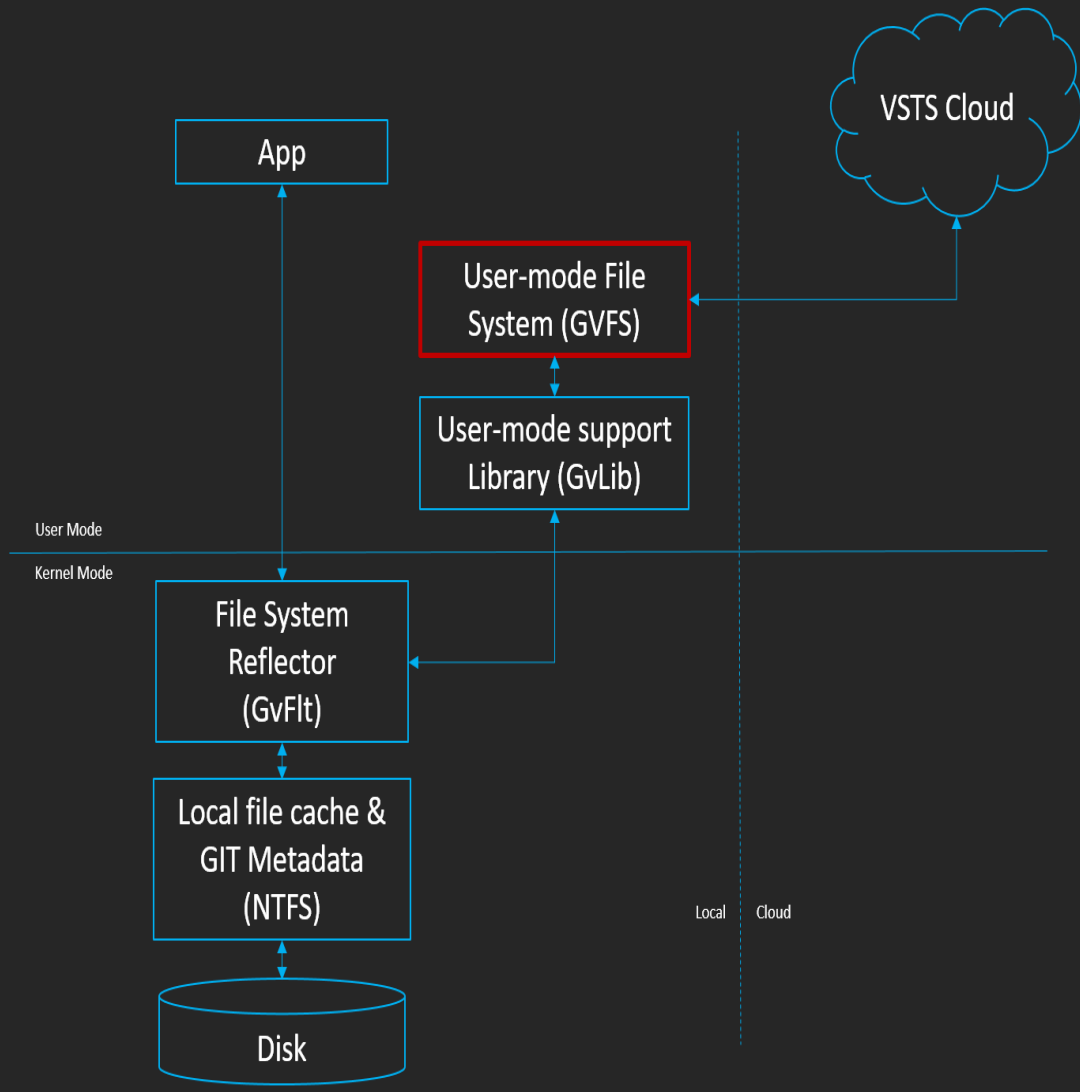
Directly reflects FS operations:

- Post Create
- Pre and Post Rename

No direct mapping to FS operations:

- Pre First Write
  - For Files: Before data modification
  - For Directories: Before creation, rename or deletion of a child item
- Pre Delete
  - PreSetFileDisposition
  - PreCleanup for handle with DeleteOnClose
- Handle Closed
  - Notification of file modified
  - Notification of file deleted

User-mode File System for Git (GVFS)



GVFS goals

- 0. Virtual repos behave like real repos in every way that matters to users
- 1. Git commands are fast on arbitrarily large repos
- 2. Don’t download unnecessary contents

Goal 0 – Behave like a normal Git repo

- Respond to GvFlt callbacks to
  - Enumerate directories
  - Hydrate files
- Building on top of a filter driver means we inherit all the rich functionality of NTFS. Build tools, etc, continue to work as before.

- This implementation also implies goal 2 – only download the contents that are accessed, thanks to the lazy nature of GvFlt

Goal 1 – Make Git commands fast

- GVFS reduces Git command run times from O(files in repo) to O(files modified by user)
- Git has a pre-existing sparse-checkout feature
- GVFS manages the set of files that Git will consider
  - The set starts out empty
  - Files that the user modifies are added to this set
- Our world view is:
  - Git owns the set of files that the user has modified
  - GVFS virtually projects everything else

GVFS goals

- Make Git commands fast
  - Git on NTFS is O(N) on the number of files in the working directory
  - With GVFS, Git becomes O(N) on the number of files you've modified
- Only download what's needed
- Make sure all Git commands function correctly
  - The end result should look the same as a normal repo on NTFS

Git perf with GVFS

~~12~~ ~~hrs~~ 90 secs **clone**

~~3~~ ~~hrs~~ 30 secs **checkout**

~~8~~ ~~mins~~ 3 sec **status**

~~30~~ ~~mins~~ 10 secs **commit**

Cache layers

- Internet
  - Source of truth: a server such as VSTS
- Intranet
  - Cache servers: machines on the LAN with copy of all data
- Local machine
  - Git object cache: Compressed Git objects are stored in a volume-wide cache, shared by all repos on that volume
  - Working directory: Uncompressed contents are cached in placeholder files once a user reads their contents

Next steps

- Continue to optimize performance of various Git commands
  - e.g. ‘git status’ is now in the 3-5s range, but we want <1s
- Optimize our caching and prefetching strategies
  - Git command performance is highly sensitive to misses on commit and tree objects
  - e.g. a command that should take <10s can end up taking minutes
- GVFS for Mac