Lecture 10: Version Control and Backups

1. Introduction: Why Version Control?

- **Version control** is about more than just tracking code changes; it's fundamentally about managing history and enabling recovery from mistakes or failures.
- **Backups** are a general concept: you want to know not only what you have today, but what you had yesterday, last week, last year, etc.
- **Version control** can be seen as a specialized backup system for source code, but backups are needed for many things beyond code.
- **Analogy:** Backups are like a time machine for your entire house; version control is a time machine for your work desk, letting you see every change you made to your projects.

2. Backups vs. Version Control: Roles and Responsibilities

- **Traditional division:**
  - *Developers*: Innovate, create new things, push the envelope.
  - *Operations staff*: Maintain stability, ensure nothing is lost, manage backups.
- **DevOps approach:**
  - Blurs the line between development and operations; everyone is responsible for both innovation and maintenance.
  - DevOps teams manage their own backups and deployments, streamlining processes.
- **Key Difference:**
  - Backups focus on disaster recovery and data retention.
  - Version control focuses on tracking changes, collaboration, and project history.

Summary Table: Backup vs. Version Control

Aspect	Backup System	Version Control System
Primary Purpose	Disaster recovery, data retention	Track changes, enable collaboration
Granularity	Whole files, directories, or disks	Individual changes (commits)
History Depth	Snapshots at intervals	Complete, fine-grained history
Collaboration	Not designed for collaboration	Built-in support (branches, merges)
Recovery Model	Restore to previous state	Revert/checkout any commit
Metadata Tracked	File metadata, timestamps	Author, message, diffs, metadata
Typical Use Cases	System restore, accidental deletion	Software development, document editing

Real-World Scenario: Backup vs. Version Control

Suppose you are working on a research paper. If you accidentally delete the file, a backup system allows you to restore the last saved version. However, if you want to see how your paper evolved, compare different drafts, or collaborate with co-authors, a version control system like Git is essential. Version control lets you track every change, revert to any previous version, and merge contributions from multiple people.

3. Why Do We Need Backups? (Failure Models)

- **Purpose:** To recover from failures (hardware, user error, attacks, bugs, insider threats).
- **Failure model:**
  - You must know what kinds of failures you want to protect against to design a good backup strategy.
  - Without a clear failure model, you risk wasting resources or missing critical risks.

Common Failure Types and Examples

Type	Description	Example
Hardware	Drive fails (flash or disk); e.g., 1–2% annualized failure rate (AFR) for flash, ~2–4% for disks.	Disk crash, SSD wear-out
User Error	Accidental deletion or overwriting of files.	`rm -rf *` in wrong directory
Attacker Action	External or internal attackers delete or modify data.	Ransomware, disgruntled employee
Software Bug	Bugs that corrupt or delete data.	Faulty update script wipes configs
Insider Threat	Trusted users (e.g., with shared accounts) tamper with data.	Admin deletes logs to hide activity

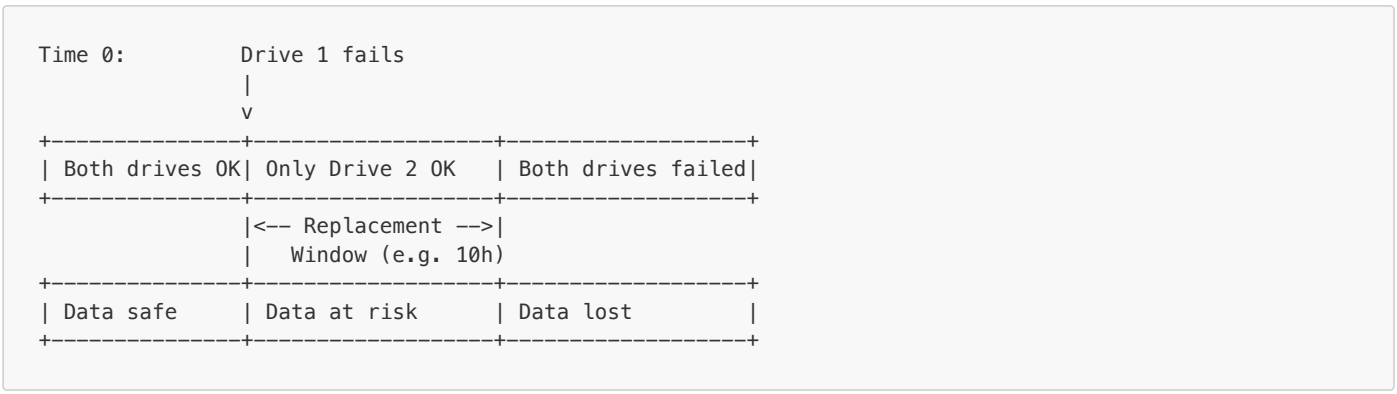
- **AFR (Annualized Failure Rate):**
  - E.g., 1–2% for flash drives in server environments; higher for disks.
  - Large installations (Amazon, Google) see these rates in practice.
  - **Correlated failures:**

    - Multiple drives from the same batch may fail together (manufacturing defect).
    - Power surges, firmware bugs, or environmental hazards can cause simultaneous failures.

4. Simple Backup Models: Mirroring

- **Mirroring:** Keep two identical copies of all data.
    - *Pros:* Simple, protects against single drive failure.
    - *Cons:* Doubles storage cost, assumes independent failures (not always true—bad batches, shared environments).
- **AFR for Mirroring:**
    - Naive: Square the AFR (e.g., 0.01 × 0.01 = 0.0001) for both drives failing in a year (if independent).
    - Realistic: Must consider correlated failures and recovery time (e.g., time to replace a failed drive).
    - Actual risk is lower if failed drives are replaced promptly, but higher if failures are correlated.
    - **Revised AFR Calculation (Considering Recovery Time):**
        - If a drive fails and is replaced within a certain time window (e.g., 10 hours), the probability of data loss is the probability that the second drive fails during that window.
        - The revised formula is:
            - `AFR ≈ AFR_single * (AFR_single * (drive replacement time / 1 year))`
            - For example, if AFR_single = 0.01 (1%), and replacement time is 10 hours:
                - `AFR ≈ 0.01 * (0.01 * (10 / (365*24)))`
        - This is much lower than simply squaring the AFR, provided prompt replacement and independent failures.

    Mirroring and Recovery Window (Detailed ASCII Diagram):



Data is only at risk if the second drive fails during the replacement window after the first failure. If both drives are in the same enclosure and a power surge occurs, both may fail at once—mirroring does not protect against this.

5. Recovery Models: Testing Your Backups

- **Don't just model failure—model recovery!**
    - Many engineers forget to test recovery.
    - Example: Simulate a drive failure by unplugging a device and see if you can recover.
    - *Personal test:* Lose your laptop for a weekend—can you still get your work done?

> **Philosophy:**
> "Don't just think it, do it." Test your recovery model in practice, not just in theory.

Common Mistakes Table:

Mistake	Consequence
Only model failure, not recovery	Backups may be useless in a real emergency
Forget to test restore process	Can't recover when needed
Assume backups work	False sense of security
Delete incrementals blindly	Lose ability to restore full history

Backup Testing Checklist:

- ☐ Can you restore a single file?
- ☐ Can you restore an entire directory?
- ☐ Can you restore to a new machine?
- ☐ Are all permissions, links, and metadata preserved?
- ☐ Are encrypted files still accessible?

6. What to Back Up?

- **Data to back up:**
  - File contents
  - File names
  - Directory structure
  - Permissions, ownership, access rights (as shown by `ls -l`)
  - Timestamps (modification time)
  - Inode/link count (for hard links)
  - Symbolic link contents and file types
  - Backup meta-information (version number, timestamp, creator)
- **Delicate files:**
  - Passwords, encryption keys—handle with care! Don't store in backups unless secured; many breaches have resulted from mishandling these.
- **Be thorough:**
  - If `ls -l` or `ls -li` shows it, you probably need to back it up.
  - Easy to forget things like hard links or symlink contents—be meticulous.

File System Metadata Table:

Metadata	Why It Matters
Permissions	Security, access control
Ownership	Determines file access
Timestamps	Auditing, synchronization
Hard links	Prevents data loss, preserves structure
Symlinks	Maintains references, avoids breakage
File types	Correct restoration

7. Backup Strategies: Periodic, Incremental, and Rotational

- **Full backup:** Snapshot of the entire system at a point in time.
- **Incremental backup:** Only changed files since the last backup.
  - *Pros:* Saves space and time.
  - *Cons:* Restoring requires all incrementals to be intact; more fragile.
- **Rotational/Consolidation:**
  - Keep frequent (e.g., hourly) backups for recent data, less frequent (e.g., daily/weekly) for older data.
  - Combine adjacent incrementals to save space.
  - *Caution:* Don't delete incrementals blindly—combine them to avoid losing the ability to restore.

Backup Rotation/Retention Flow:

As time passes, frequent backups are consolidated or deleted, retaining only less frequent snapshots for older data.

Scenario:

- You have hourly backups for the last 24 hours, daily for the last month, and monthly for the last year. If you need a file from 6 months ago, you restore from the monthly backup.

8. Efficient Backup Techniques

- **Differencing (diff/patch):**
  - Store only the changes (diffs) between file versions, not the whole file.
  - Example:

```
diff -u old_file new_file > changes.diff
patch old_file < changes.diff
```

  - *Pros:* Saves space, especially for large files with small changes.
  - *Cons:* All diffs must be intact to restore; more fragile.
- **Data grooming:**
  - Don't back up files you can easily recreate (e.g., `.o` files, build artifacts).
  - Use scripts to exclude such files before backup.
  - *Note:* Data grooming often requires user input or customization to avoid missing important files.
- **Deduplication:**
  - Avoid storing identical data blocks multiple times.
  - Block-level deduplication: Only unique blocks are stored; identical blocks are referenced by pointers.
  - *Copy-on-write (COW):* New blocks are only created when data is changed.
  - *Caveat:* If a shared block is lost, all referencing files are affected. Not a substitute for true redundancy—always maintain physically separate copies for critical data.
- **Compression:**
  - Use lossless compression (e.g., gzip) to reduce backup size.
  - *Tradeoff:* Slower backup/restore, but often negligible with modern CPUs.
- **Encryption:**
  - Encrypt backups to protect against unauthorized access (especially if stored offsite or in the cloud).
- **Staging (multi-tier backups):**
  - Use multiple backup layers: e.g., flash → disk → tape.
  - Tape is slowest/cheapest, disk is mid-tier, flash is fastest/most expensive.
- **Multiplexing:**
  - Use a single backup device for multiple data sources to save on hardware costs.
- **Checksums:**
  - Compute and store checksums (e.g., SHA256) for backup data.
  - Later, verify backups by comparing checksums—cheap and effective for integrity checking.
- **Format conversion:**
  - Use lossless compression (e.g., gzip) to reduce backup size.
  - *Note:* Format conversion (e.g., CRLF vs. LF) is often a pain, but Git can handle it automatically for you, saving you from manual headaches.

Technique Comparison Table:

Technique	Space Savings	Speed Impact	Risk/Tradeoff
Deduplication	High	Low	Shared block loss affects many files
Compression	Medium	Low/Medium	Slower restore, but usually minor
Encryption	None	Low/Medium	Key management is critical

9. Caching and Backup Reliability

- **Cache vs. backup:**

- Caches are small/fast, backups are big/slow.
- Don't rely on caches as backups; treat them as primary data, not backup copies.

- **Incremental/diff risk:**
  - If any incremental or diff is lost/corrupted, you may lose the ability to restore.
  - Reliability of storage is critical for incremental/diff-based backups.

  Scenario:
  - Your browser cache lets you quickly reload a page, but if your hard drive fails, the cache is lost. Only a backup can restore your data.

10. From Backups to Version Control: Developer Needs

- **Backups alone are not enough for software development.**
- **Version control systems (VCS) provide:**
  - Efficient lookups of old versions (must be fast, not just possible).
  - File rename tracking: Recognize when a file is renamed, not just deleted/created.
  - Metadata about development history (e.g., tags, commit messages, authorship).
  - Atomic commits: Group multiple changes into a single, all-or-nothing operation. Prevents inconsistent intermediate states.
  - Hooks: Custom scripts (pre-commit, post-commit, etc.) to enforce policies or automate tasks.
  - Format conversion: Handle line ending differences (e.g., CRLF vs. LF) between platforms.
  - Signed commits: Cryptographically verify authorship and integrity of changes.

  Backup vs. Version Control Table:

Feature	Backup System	Version Control System
Fast lookup of history	No	Yes
Track renames/moves	No	Yes
Metadata (commits, etc)	No	Yes
Atomic changes	No	Yes
Collaboration tools	No	Yes
Signed/authenticated	No	Yes

11. Git: The Most Popular Version Control System

- **Two key concepts in Git:**
  1. **Object database:**
     - Stores the complete history of your project as a database of objects (blobs, trees, commits, tags).
     - Persistent, shared with other developers.
  2. **Index (cache):**
     - Stores your planned changes (the "staging area").
     - Private to you, not shared until committed.
     - Allows you to prepare and review changes before committing to history.

- **Workflow:**
  - You can spend time perfecting your index (staging area), then commit to the object database for a clean, understandable history.
  - Clean histories are easier for collaborators to review and accept.

- **Git clone:**
  - Copies the object database, creates a trivial index, and sets up working files.
  - Local clones are fast and use hard links for deduplication.

  Git Data Flow (Detailed):

You edit files, stage changes in the index, and commit them to the object database (history). You can also reset or checkout to move between these states.

12. Real-World Example: CSNet Snapshots

- **Snapshot directories:**
  - Many systems (e.g., CSNet) provide snapshot directories with hourly, daily, and weekly backups.
  - Implemented using block-level deduplication and snapshotting.
  - Users can recover files from up to two weeks ago, with all metadata preserved.

  Scenario:
  - You accidentally delete a file. You navigate to the snapshot directory, find yesterday's version, and restore it with all permissions and links intact.

13. Summary: Best Practices

- Always have a clear failure and recovery model.
- Test your backups—don't just assume they work.
- Be thorough in what you back up (metadata, links, permissions, etc.).
- Use efficient, cost-effective strategies (incrementals, deduplication, compression, encryption, checksums).
- For software development, use a version control system that supports atomic commits, metadata, hooks, and efficient history access.
- Understand the distinction between backups and version control, and use both appropriately.

Best Practices Checklist:

- ☐ Have a documented failure model
- ☐ Regularly test recovery
- ☐ Back up all necessary metadata
- ☐ Use efficient backup strategies
- ☐ Use version control for code and documents
- ☐ Secure sensitive data in backups
- ☐ Periodically review and update backup/versioning procedures