**GIT/GITHUB  
PART-1  
Chapter 1: Managing Change in IT with Version Control**

In IT, scripts and configuration files constantly change as systems grow or requirements shift. Tracking these changes is essential.

**Challenges:**

* Frequent updates to scripts and settings
* Hard to track what changed, when, and why
* Quick fixes can cause bugs
* Undoing changes without history is risky

**Why Version Control?**

* Tracks all changes over time
* Eases troubleshooting
* Enables safe rollbacks
* Reduces errors
* Supports collaboration

**Example:**  
A new script checks firmware but fails on some systems. A quick fix introduces more bugs. With version control, you can safely roll back, fix properly, and test before redeploying.

**Git Basics:**

* Git is a version control tool used via the command line
* Tracks changes in scripts, config files, and documents
* Tools like diff and patch help compare and apply changes

**Chapter 2: Primitive Version Control and Its Evolution**

Have you ever saved backup copies of your work or emailed updates to teammates? If so, you've used a basic form of version control—keeping historical copies.

**What It Looked Like:**

* Manually saving multiple versions
* Sharing updates via email
* Keeping backups before major deletions
* Useful for going back or tracking progress

**Limitations of Primitive Version Control:**

* Must remember to create backups
* Copies the entire project even for small changes
* Difficult to track who made changes or why

**Why Version Control Matters:**

* Tracks changes in code, images, configs, and more
* Makes collaboration easier
* Helps understand the "what" and "why" behind changes

**Next Step:**  
Git provides a smarter, automated way to handle version control. It improves collaboration, change tracking, and historical record-keeping—all without the chaos of manual backups.

**Chapter 3: Comparing Code with diff and Other Tools**

When working with different versions of a file, identifying what has changed can be difficult and time-consuming if done manually. Thankfully, some tools make this process efficient and accurate.

The Better Way: Use Comparison Tools

Instead of manually checking for differences, use tools designed to highlight changes. The most common tool for this on the command line is diff.

How the diff Command Works

The diff utility compares two files line by line and displays the changes between them.

* Lines removed are shown with <
* Lines added are shown with >

Common Output Format

* 5c5,6: Line 5 in the first file was changed and replaced by lines 5 and 6 in the second file.
* 11a13,15: Three lines were added after line 11 in the first file.

Unified Format with diff -u

Using the -u (unified) flag provides better context for the changes:

* Lines removed are prefixed with -
* Lines added are prefixed with +
* Unchanged lines are shown to give context around the changes

This format is especially useful when reviewing modifications to more complex code blocks such as functions, conditionals, or loops.

Other Useful Tools

In addition to diff, several other tools can help compare files more effectively:

* wdiff – Compares files word by word instead of line by line.
* GUI Tools (graphical user interface):
  + Meld
  + KDiff3
  + vimdiff

These provide side-by-side comparisons with colour highlighting, making it easier to visually identify changes.

**Chapter 4: Using diff and patch to Share and Apply Code Changes**

When collaborating on code, explaining a bug fix in words alone can be confusing, especially for complex scripts. Instead, it’s much clearer and more efficient to share the changes using a **diff file**.

**Creating a Diff File**

To show the difference between two versions of a file, use the diff command with the -u flag (for "unified" format):

diff -u original\_file updated\_file > changes.diff

* The -u option adds helpful context to the diff output.
* The > symbol redirects the output into a file named changes.diff, which can be shared.

This file contains the exact lines that were added, removed, or modified, along with context lines that make the change easier to understand.

**Applying Changes with patch**

To apply the changes from a diff file to the original file, use the patch command:

patch target\_file < changes.diff

* patch reads the diff file and automatically updates the target file with the specified changes.
* This automates what would otherwise be a manual and error-prone process.

**Why Use Diff and Patch?**

There are several advantages to using this approach:

* **Clarity:** Only the changes are shared, not the entire file, making reviews easier.
* **Version Handling:** If the recipient’s file is slightly different from yours, patch may still apply the changes correctly by using context lines.
* **Efficiency:** Especially useful when modifying multiple files in large projects. Directory-level diffs can specify changes across different folders without requiring full file replacements.

**Chapter 5: Debugging and Fixing a Script Using diff and patch**

Imagine a situation where a colleague asks for help fixing a broken script.   
  
**Step 1: Create Copies of the Script**

Before making any changes, create two copies of the script:

* **\_original**: This is the unmodified version, kept for comparison.
* **\_fixed**: This will be the version you will modify to fix the issues.

**Step 2: Identify the Bugs in the Script.**

**Step 3: Apply the Fixes**

**Step 4: Generate a diff File**

Now that the script is fixed, it's time to share the changes with your colleague. You can generate a diff file, which shows the differences between the original script (\_original) and the fixed script (\_fixed).

To create the diff file, you can use the diff -u command, which will output the differences in a unified format, providing context to understand the changes:

diff -u disk\_usage\_original.py disk\_usage\_fixed.py > disk\_usage.diff

The resulting diff file will include the changes necessary to fix the bugs.

**Step 5: Apply the Patch. Using the patch**

To apply the changes, your colleague can use the patch command with the generated diff file. They can run the following command:

patch < disk\_usage.diff

This command will automatically apply the necessary changes to their version of the script.

**Step 6: Verify the Fix**

Once the patch is applied, they can execute the script and verify that it runs correctly without errors.

**ONE SHOT PART 1:**  
**Version Control and Debugging Using diff and patch**

* **Managing Change in IT**: Version control helps manage frequent script/config changes by tracking changes, easing troubleshooting, enabling rollbacks, reducing errors, and supporting collaboration.
* **Primitive Version Control**: Early version control methods like manual backups and email updates had limitations like lack of tracking and collaboration. Git automates version control, improving tracking and collaboration.
* **Comparing Code with diff**: Tools like diff compare code and highlight changes. The -u (unified) flag gives better context for changes. Other tools like wdiff and GUI-based tools (Meld, KDiff3, vimdiff) provide word-level and side-by-side comparisons.
* **Using diff and patch**: diff -u creates a file with the differences between two versions of a file, and patch applies the changes. This method is more efficient than manual changes and helps maintain version control.
* **Debugging and Fixing Scripts**: When fixing a script, create copies for comparison, identify bugs, and apply fixes. Use diff -u to generate a diff file with changes, which can be shared and applied using the patch command to update the script automatically and verify the fix.

**PART-2**

**Chapter 6: Understanding Version Control Systems (VCS)**

**What is a Version Control System?**

* A **Version Control System (VCS)** tracks changes made to files over time.
* It records **who** made changes, **when** they were made, and allows easy **reversion** if necessary.

**Benefits of a VCS**

* **Collaboration:** Multiple contributors can merge changes from different sources.
* **History Tracking:** Keeps all past versions, unlike regular file systems.
* **Structured Changes:** Groups multiple file edits into a single **commit**.
* **Commit Messages:** Authors can document **why** changes were made, referencing issues or bug fixes.
* **Debugging Aid:** Helps trace back through the change history to understand or fix issues.

**Repositories**

* Code and files are organised into **repositories**.
* A repository can be used by **a single person** or **thousands of contributors**.
* Contributors may have access to specific repositories depending on their roles.

**Beyond Code**

* A VCS isn't limited to source code; it can store:
  + Configuration files
  + Documentation
  + Data files
  + Any other content that needs tracking

**Limitations**

* Works best with **text-based files** due to compatibility with tools like diff and patch.
* **Binary files** (like images or videos) can be stored, but:
  + Differences between versions are harder to track
  + Automatic merging may not be possible

**Why Use a VCS Instead of Backups?**

* VCS offers **structured, automated tracking**, collaboration tools, and rich history—all of which backups alone cannot provide.

**Chapter 7: Why Use a VCS as a Solo IT Specialist?**

**1. Long-Term Clarity and Accountability**

* Even if you're the only one coding or writing scripts, a VCS stores **both your code and its history**.
* It acts like a **time machine**, helping you recall **why** changes were made, especially when documented with clear **commit messages**.
* This prevents confusion when revisiting old scripts or configurations after weeks or months.

**2. Real-World Examples & Benefits**

**📁 Example 1: DNS Zone Files**

* A DNS zone file maps hostnames to IPS in your network.
* Storing it in a VCS lets you:
  + Track **when and why** each entry was added.
  + View or **revert to previous versions** if something breaks.
  + Fix problems faster thanks to the **audit trail**.
  + Focus on recovery first (“stop the bleeding”), debug later.

**📁 Example 2: DHCP Server Configurations**

* When you use DHCP with a **failover setup**, both primary and standby servers need **identical config files**.
* VCS helps by:
  + Storing a **single source of truth** for the configuration.
  + Ensuring all servers **sync to the same version**.
  + Making rollbacks easy in case a bad change (like a duplicate entry) causes failure.
  + **Automating** new server deployment using stored configuration files.

**3. Key Takeaways**

* A VCS boosts system **reliability** and **scalability**.
* It allows for **faster recovery**, **better documentation**, and **automated deployments**.

**Chapter 8: What Is Git and Why Is It So Popular?**

**🧑‍💻 Created by Linus Torvalds (2005)**

* Linus, the developer behind the **Linux kernel**, created Git to handle its development challenges.
* He needed a **fast, distributed, and flexible** version control system, which existing tools couldn't provide—so he built his own.

**🌍 Git's Key Features**

* **Free and Open Source**: Works on Unix/Linux, Windows, and macOS.
* **Distributed Architecture**: Every contributor has a **complete copy** of the repository, not just the latest version.
* **No Internet Needed**: Most Git operations are **local and fast**, since you're working with your own full copy of the repo.
* **Collaboration-Ready**: Although Git doesn’t need a central server, setting up a **shared repo** on a server helps teams sync changes.
* **Flexible Use**:
  + Use Git as a **standalone tool** on a local machine.
  + Host a **server** to share a repo with others.
  + Access it from another machine as a **client**.
* **Protocol Options**: Supports **HTTP, SSH**, and Git’s own protocol for communication over a network.

**🌐 Git's Scope & Use Cases**

* Ideal for:
  + **Solo developers** working offline.
  + **Large teams** collaborating across geographies.
  + **Private repositories** or **open-source projects** hosted on platforms like **GitHub**, **GitLab**, etc.

**📖 SCM vs VCS**

* Git's official site is [git-scm.com].
* **SCM = Source Control Management**, another term for **Version Control System (VCS)**.
* While the terms are often interchangeable, **VCS** is preferred since these systems manage more than just source code.

**🛠️ Alternatives Exist**

* While Git is widely used, other tools include:
  + **Subversion (SVN)**
  + **Mercurial**
* Feel free to explore those if Git doesn’t fully meet your needs.

**✅ Why Git Was Chosen for This Course**

* **Popularity**
* **Cross-platform support and robust.**

**ONE SHOT PART 2  
Essentials of Version Control Systems & Git (Ch. 6–8)**

* **VCS tracks changes** to files with history, authorship, and easy rollback.
* Enables **collaboration**, even for solo users (clarity, recovery, audit trail).
* Stores more than code: configs, docs, data – ideal for IT specialists.
* **Git**, created by **Linus Torvalds** in 2005, is free, fast, and distributed.
* Works **offline**, supports **HTTP/SSH**, and runs on all major OSes.
* Git doesn’t require a central server but supports **collaboration platforms** (e.g., GitHub).
* Preferred for its **speed, flexibility, and cross-platform support**.
* Alternatives: **SVN**, **Mercurial**, but Git remains the most popular.

**PART-3**

**Chapter 9: Getting Started with Git – Key Concepts & Setup**

**1. Git Configuration**

Before using Git, tell it who you are:

git config --global user.name "Your Name"

git config --global user.email "your.email@example.com"

* --global applies this configuration to **all repositories**.
* You can also set user info per repository (without --global).

**2. Creating or Cloning a Repository**

* **Create new repo**:
* git init
  + Initialises an **empty Git repository** in the current directory.
  + Creates a hidden folder called .git, which holds Git’s **internal data** (the database of changes).
* **Clone existing repo**:
* git clone <repository\_url>
  + Copies the .git directory and working files from a remote location.

**3. Git Directory vs Working Tree**

* **.git/ (Git directory)**: Internal Git data like history and configurations.
* **Working Tree**: Current project files visible to you — your working copy.

**4. Tracking Files**

* By default, new files are **untracked**.
* To track a file:
* git add <filename>
  + This adds it to the **staging area (aka index)** — a place to prepare files for committing.

**5. Check Git Status**

git status

* Shows:
  + Tracked and untracked files
  + Which files are staged and ready to commit

**6. Commit Your Changes**

git commit

* Opens a text editor (e.g., nano) to enter a **commit message**.
* Saves the staged changes to the Git directory as a **commit** — a snapshot in time.

**🔁 Summary of Git File Lifecycle:**

1. File created → **untracked**
2. git add → **staged**
3. git commit → **committed** (saved in .git)

**Chapter 10: How Git Tracks File Changes: A Snapshot-Based Workflow**

* **Git Project Structure:**
  + **Git Directory**: Stores snapshots (commits) — complete history.
  + **Working Tree**: Your current project files.
  + **Staging Area**: Holds changes marked for the next commit.
* **Snapshots = Commits**:  
  Git captures the state of your project at each commit like a photo.
* **File Tracking States:**
  + **Untracked**: New files not yet added to Git.
  + **Tracked**: Files under Git’s control, in one of three states:
    - **Modified**: File changed but not staged.
    - **Staged**: Changes marked using git add.
    - **Committed**: Staged changes saved to history with git commit.
* **Basic Workflow:**
  + Modify a file → Becomes **modified**.
  + Run git add <file> → File is **staged**.
  + Run git commit -m "message" → Changes are **committed** as a new snapshot.
* **Command Tips:**
  + git status → Shows current file states.
  + git add → Moves changes to the staging area.
  + git commit -m "msg" → Saves a snapshot.
* **Cycle Recap**:  
  Modified → Staged → Committed (stored in Git directory).

**Chapter 10: Git Project Structure Recap**

Every Git project has three main sections:

1. **Git Directory** – Stores all commits (snapshots), branches, and version history.
2. **Working Tree** – The current state of the project you’re working on.
3. **Staging Area** – Where you place changes you want to include in your next commit.

**🔄 File States in Git**

Each file can be in one of the following states:

* **Untracked**: New files not added to Git yet.
* **Modified**: Tracked files that have changed but are not yet staged.
* **Staged**: Files that are ready to be committed.
* **Committed**: Changes stored as a snapshot in the Git directory.

**🛠️ Basic Git Workflow**

1. **Initialize Repository**
2. git init

Creates a new Git repository.

1. **Check Configuration**
2. git config -l

Shows user.name, user.email, and other config settings.

1. **Create/Edit Files**
   * Start with writing/editing your code/script.
   * Example: Create all\_checks.py with a basic structure.
2. **Check Status**
3. git status
   * Shows if files are untracked, modified, staged, or clean.
4. **Track or Stage Changes**
5. git add filename.py
   * Moves files from **untracked** to **staged**, or from **modified** to **staged**.
6. **Commit Changes**
7. git commit -m "Descriptive commit message"
   * Saves a **snapshot** with a message in the Git directory.

**🧾 Tips:**

* Use git status frequently to track file states.
* Practice makes perfect — repeat the modify → add → commit cycle to build familiarity.
* Use git config to manage your identity and editor preferences.

**Chapter 11: What Makes a Good Commit Message?**

1. **Think of the reader** (including future you!):
   * Include *why* the change was made, not just *what*.
   * Add links to relevant design docs or issue trackers if helpful.
2. **Structure of a Commit Message**:
   * **First line (summary)**: Short and clear — usually ≤ 50 characters.
   * **Blank line**: Separates the summary from the details.
   * **Body (description)**: Explains the change in more detail, wrapped at 72 characters per line.

**📜 Viewing Commit History: git log**

* Use git log to view:
  + **Commit ID** (a unique hash string)
  + **Author** (name and email)
  + **Date and time**
  + **Commit message**
* The first listed commit often shows HEAD -> master, which means it's the latest commit on the master branch.
* The log output can be long if the messages aren't wrapped properly — that's why line limits help.

**ONE SHOT PART 3:**  
**Git and basic Git commands**

* git config --global user.name/email sets identity
* git init creates new repo; git clone <url> clones one
* .git/ stores history; working tree is current files; staging area preps files
* New files are untracked → use git add to stage → git commit -m "msg" to save
* File states: untracked → modified → staged → committed
* git status shows file states
* Commit message: short (≤50 chars) summary, blank line, detailed body (wrap at 72 chars)
* git log shows commit hash, author, date, and message