GIT/GITHUB

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.