Hello. It told me about two new things. The second one was "Agent" and I chose it. Do you know what the first might have been?

ChatGPT said:

Worked for 2 seconds

I’m sorry, but I’m not aware of what the first option was. Could you tell me more about it?

You said:

I think it said something about connecting apps. I don't want to set up the connection right now, but I do want to have the option.

ChatGPT said:

Worked for 1 minute

It sounds like the first option you saw was probably the new **“Connect your apps”** feature (sometimes called “Use connectors”). Connectors let ChatGPT link to third-party services such as Google Drive, GitHub or SharePoint so it can search your files, pull live data and reference that content in chathelp.openai.com. To add one, you go to **Settings → Connectors → Connect**, which sends you through the chosen service’s normal login/authorization flowhelp.openai.com. You don’t have to enable this now—in fact, you can skip it entirely until you decide it would be useful. When you’re ready, simply add a connector and ChatGPT will then be able to use that app’s data in your chats.

You said:

Can I return to the standard GPT prompt while still retaining access to agent mode when I want it?

ChatGPT said:

Stopped after 1 minute

You said:

Am I back to the standard ChatGPT prompt?

ChatGPT said:

Yes — by default you’re back in the standard ChatGPT prompt. Agent mode isn’t “always on”; it’s something you explicitly switch into when you want to use it (for example, via the **Tools menu** or the **/agent** command, depending on your interface). You can continue normal conversations in the regular ChatGPT prompt, and when you need Agent features, just turn it on. Then you can switch right back to the standard mode afterward.

Would you like me to walk you through exactly how to toggle between the two in your interface?

You said:

Nope, I'll look that up later. I'm simply starting a new chat to reduce lag time.

ChatGPT said:

Got it — starting fresh chats is a good way to cut down on lag. You'll be in the standard ChatGPT prompt by default in any new thread, and agent mode will only come into play if you explicitly switch it on.

Want me to also keep answers lean and quick in this thread to match your goal of reducing lag, or do you prefer the usual full detail?

You said:

I prefer the full detail. I can start a new chat if the lag gets too much. I have a context markdown file that I'll share with you.

ChatGPT said:

Perfect — full detail it is. If you share your context Markdown file, I'll read through it carefully and use it to stay aligned with your goals, preferences, and style. That way you won't have to repeat background information each time.

Go ahead and drop the file when you're ready.

You said:

Before I do so, I have a question. I was told by someone I admire that it's often better to use the shebang, #!/usr/bin/env bash over #!/bin/bash. That makes a lot of sense if the OS in question doesn't have a /bin/bash, but rather a /some/other/path/bash. This friend is not omniscient (though I know you aren't either, and I sure as hbar (my physics-based replacement for 'hell') am not. The first draft you gave me has just the #!/bin/bash shebang. What are your thoughts on that vs. #!/usr/bin/env bash. I think for some reasom, that the usr stands for Uniform System Resources or maybe Universal System Resources. Is it standard on *NIX/*nux systems to use /usr/bin/env?

ChatGPT said:

Great question — this is one of those "Unix traditions" where the details matter. Let me break it down for you:

/bin/bash vs. /usr/bin/env bash

- **#!/bin/bash**
    - Hardcodes the path to Bash as /bin/bash.
    - Works fine on systems where Bash is guaranteed to live there (common on many Linux distributions).
    - Problem: not universal. Some systems (notably BSDs, macOS, or minimal container images) might install Bash in a different location, such as /usr/local/bin/bash.
- **#!/usr/bin/env bash**
    - Invokes Bash via the env program, which looks up bash in your PATH.
    - This makes the script portable across systems where Bash is installed in different places.
    - The downside is that you rely on the user's PATH being correctly set — if their PATH points somewhere unexpected, env will run whatever "bash" it finds.

What /usr stands for

/usr doesn't mean *Uniform* or *Universal* — it originally meant **"user system resources"**, and over time it became the standard directory for user-level binaries and libraries across UNIX systems.

- So /usr/bin/env is standard on essentially every UNIX-like system (*nix/*nux, BSD, macOS, Linux).
- env itself is part of the POSIX toolset, and its location at /usr/bin/env is one of the most portable assumptions you can make.

Common Practice

- For **maximum portability**, #!/usr/bin/env bash is generally preferred, especially if you plan to share scripts across multiple environments.
- For **controlled environments** (e.g., Linux servers where you know Bash is in /bin/bash), #!/bin/bash can be slightly faster and more predictable.

Special Cases

- If you’re writing scripts that only use POSIX shell features, it’s even better to use:
- #!/bin/sh

since /bin/sh is guaranteed by POSIX.

- But if you need Bash-specific features (arrays, [[ ]], ** globbing, etc.), stick with bash.

✅ **Bottom line**:
Use #!/usr/bin/env bash if you care about portability across unknown systems.
Use #!/bin/bash if you’re targeting a known Linux environment where you can guarantee Bash’s location.

Would you like me to also show you some **edge cases** (like when /usr/bin/env bash can *fail*), so you can decide whether it’s worth the tradeoff for your scripts?

You said:

I would like to know about edge cases where /usr/bin/env bash can fail, but I'd like you to give me a somewhat-detailed rundown at which I will look later. I like the "Bourne Again" features of bash (vs. plain sh, which I don't know to be even used on some systems), and I've avoided problems when working with colleagues using other (mac, BSD) OSes by using #!/usr/bin/env bash (as well as #!/usr/bin/env python, perl, etc.) I'll keep your notes for further review or reference (when something breaks with the env option).

ChatGPT said:

Thought for 19s

Absolutely—here’s a compact-but-detailed rundown you can stash for later. It’s focused on when #!/usr/bin/env bash can bite, plus quick mitigations.

When #!/usr/bin/env bash can fail (or surprise you)

1) PATH is empty/minimal (cron, systemd, launchd, CI)

- **Symptom:** env: bash: No such file or directory or it finds the *wrong* bash.
- **Why:** Non-interactive jobs often run with a tiny PATH (e.g., /usr/bin:/bin), missing where your preferred Bash lives (e.g., /usr/local/bin).

- **Mitigate:**
    - In the job config, set a full PATH (cron PATH=…, systemd Environment=PATH=…, CI job env).
    - Or in the script, early line: export PATH="/usr/local/bin:/usr/bin:/bin:$PATH" (before relying on external tools).

2) Multiple Bashes; you get the "wrong" one

- **Symptom:** Features/bugs don't match expectations (e.g., macOS's /bin/bash is 3.x; Homebrew is 5.x).
- **Why:** env picks the first bash in PATH, which might be an old system copy.
- **Mitigate:**
    - Put your preferred Bash earlier in PATH (e.g., Homebrew's /usr/local/bin or /opt/homebrew/bin).
    - If you *must* pin a version, consider an absolute shebang to that version in controlled environments.

3) Host without /usr/bin/env

- **Symptom:** No such file or directory on embedded/minimal systems (some BusyBox images, ultra-slim containers, unusual UNIXes).
- **Why:** Assumption that env is at /usr/bin/env is *very* common, but not 100% universal (sometimes it's /bin/env).
- **Mitigate:**
    - Check target images; if necessary, use #!/bin/env bash **or** install/symlink env to /usr/bin/env.
    - For truly tiny images, you may be better off with a guaranteed absolute interpreter path.

4) Early-boot contexts (before /usr is mounted)

- **Symptom:** Script used in initramfs/early boot fails to start.
- **Why:** /usr might be a separate filesystem not yet mounted, so /usr/bin/env isn't available.
- **Mitigate:** Use a shell guaranteed to exist in the initramfs (often /bin/sh or /bin/bash) for early-boot scripts.

5) Security-sensitive contexts (PATH hijacking)

- **Symptom:** Malicious/accidental bash earlier in PATH gets executed.
- **Why:** env obeys PATH. In risky contexts (shared machines, writable directories in PATH), that's dangerous.

- **Mitigate:**
    - Use an absolute interpreter (#!/bin/bash) when you *control* the deployment host and trust that path.
    - Or sanitize PATH at the top of the script:
      export PATH="/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin"

6) Shebang argument quirks (passing flags)

- **Symptom:** #!/usr/bin/env bash -e doesn't behave portably; some systems only pass one arg, some none.
- **Why:** Shebang parsing is OS-specific. Historically you couldn't reliably pass multiple arguments via env.
- **Mitigate:**
    - Prefer moving options into the script body:
    - #!/usr/bin/env bash
    - set -euo pipefail
    - Newer env supports -S (split args), e.g.,
      #!/usr/bin/env -S bash -euo pipefail
      …but **not** all platforms have -S. Use only if you control the target fleet.

7) Setuid/setgid scripts (generally discouraged)

- **Symptom:** OS ignores setuid bit on scripts or security policies block them; PATH resolution would be unsafe anyway.
- **Why:** Using env with privileged scripts magnifies risk (PATH-based resolution).
- **Mitigate:** Avoid setuid scripts; use a small compiled wrapper or a service manager.

8) Interactive vs. non-interactive differences

- **Symptom:** Works in your terminal, fails in GUI apps/launch agents.
- **Why:** GUI sessions often have different PATH than your shell init files.
- **Mitigate:** Don't rely on shell startup files for PATH; set it where the job runs or inside the script.

9) Performance nitpicks

- **Symptom:** Micro-overhead when mass-spawning thousands of short-lived scripts.
- **Why:** env adds one extra exec/lookup. Usually negligible; only matters at extreme scale.
- **Mitigate:** In hot paths on homogeneous hosts, prefer absolute shebangs.

10) Non-Bash analogues (Python/Perl/Ruby)

- **Symptom:** Virtualenv/pyenv/rbenv/conda expectations aren't met in non-interactive contexts.
- **Why:** The right shim/venv may not be in PATH where the script runs.
- **Mitigate:** Ensure the job's PATH activates the intended environment, or call the absolute interpreter from that env.

Quick decision guide

- **Portable across unknown hosts?** Prefer #!/usr/bin/env bash.
- **Controlled fleet with known layout?** #!/bin/bash (or your known absolute path).
- **Need flags?** Put them in the script (set -euo pipefail), or use env -S *only* if you know it's supported.
- **Anything scheduled/automated?** Explicitly set PATH in the runner and/or at the top of the script.

If you want, send the script later and I'll annotate the shebang + top-of-file boilerplate for a couple of target environments (your mac, a typical Ubuntu server, and a minimal container).

You said:

Awesome. Okay, here comes the context/continuation markdown. I do have one request for you to put into your long-term memory. I like two spaces as my standard, next-lowest-level marker in Python. I also like to have a heredoc-style docstring. You've done this before, but I want it for any class, method, etc.&mdash;anything with a def, even if it's a short-and-simple example. (Here's some one-shot prompting, but something I'd like you to have in your long-term memory) What you've given before

ChatGPT said:

Understood 👍

I'll apply those Python preferences going forward in this conversation (and in future ones if you remind me):

- **Indentation:** two spaces instead of the more common four.
- **Docstrings:** always add a heredoc-style docstring (""" ... """) for any def or class, even for short examples.

When you’re ready, go ahead and drop your context/continuation Markdown file, and I’ll read it in carefully before we proceed.

You said:

Oops, I didn't give the example, but you figured it out :) Here comes the markdown.

ChatGPT said:

Great — drop the Markdown file whenever you’re ready, and I’ll read it through carefully so I can work with your context.

You said:

# Lab Notebook — bballdave025 ## Session Summary ### Focus - Scaffolding automation scripts for creating project directory structure and placeholder files. - Ensuring cross-platform support (Windows CMD, PowerShell, Bash). - Transition plan: Local prototyping first (30-min scope), then AWS deployment for full scope. ### Key Progress 1. **Windows CMD / BAT (Option 2 implemented)** - Created py_touch.py outside loops to avoid script duplication. - BAT script iterates over tags to generate directories and placeholder files. - Debugging simplified vs. ~2017 BAT practices. 2. **PowerShell & Bash equivalents** - Drafted for easier maintainability and modern workflow assumptions (since PowerShell/bash are standard). - Preserves file extensions while appending tags (e.g., README_p_02_02.md → README.md). 3. **Test Runs** - Scripts not yet verified with sample tags. - tree output will be used to confirm correct directory structure and file naming. ### Reflections - CMD-only support is less relevant in 2025, since PowerShell/bash are widely available. - Decision: prefer PowerShell/bash for future development; CMD fallback only if explicitly needed. - Plan to discuss complete scope of the project plan to help direct goals. - The idea is that this will help to avoid spending too much time on a specific part - Help ensure productivity in line with ADHD/OCD therapy goals. - ADHD/OCD productivity strategy: *Done-over-perfect*, timed 20–30 minute pushes, then iterate on AWS. - MOST IMPORTANT for today (Saturday) is *Done-over-perfect* sprint mode. ### Next Steps - **Today**: 30-minute local push (finish scaffolding, push to GitHub). - **Sunday (before 09:00 + 13:00–18:00)**: AWS push (ready ResNet-50++ experiments). - **Mon/Tue**: Draft technical vision outline for computer vision paper. - **After Tue**: Focus shifts to Chain-of-Thought ideas + interview portfolio prep. --- ## Important scripts ### structure script (PowerShell or bash) #### Powershell, file structure.ps1 What we want to try in the 30-minute session.

(PowerShell script,

`structure.ps1`

given upon request

)

#### Bash, file structure.sh What will most-likely be used on AWS, though possibly without the datasets dir.

(bash script,

`structure.sh`

given upon request

)

#### Minimal Test Runs *PowerShell*

powershell

```
# Minimal test with two tags

# Ensure working directory is where the directory structure is desired

$RootTest = "$PWD/test_project"

.\structure.ps1 -RootDir $RootTest -Tags "p_01" "p_02"


# Check structure

Get-ChildItem -Recurse -Force $RootTest
```

<h3>Dave's questions</h3> - Does PowerShell have a tree command? - How likely will it be that we need to use something akin to

powershell

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> powershell.exe -ExecutionPolicy RemoteSigned
```

bash

bash

```
chmod +x structure.sh
```

```
# Minimal test with two tags

# Once again, ensure working directory is where the directory structure is desired

ROOT_DIR="$PWD/test_project_bash"

./structure.sh "$ROOT_DIR" p_01 p_02


# Check structure

tree -a -F "$ROOT_DIR"
```

#### Expected tree-type output

(Expected `tree` output,

`test_project_tree.txt`,

given upon request.

)

### Build Model script (TensorFlow 2) Probably just the method inside a Jupyter Notebook cell (*Done-over-perfect* sprint). Later, perhaps, a changeable version for both Vanilla CNN and ResNet-50++.

(Python imports and method,

`build_model(<params>)`,

as

`build_model.py`

given upon request.

)

## Other things discussed and scripts/notebooks drafted - VERY IMPORTANT: GradCAM / guided backprop / SmoothGrad integration - IMPORTANT logging FLOPs, memory, and an estimate of cost on AWS - Future-proof the CSVs for these when we get to AWS - These will all be done, remembering *Done-over-perfect*, on AWS Benefits: 1. Parameterized filters and dense units → easy ablation / hyperparameter sweeps 2. Optional

conv blocks → ablation studies without rewriting the model 3. Multi-label support via sigmoid → supports overlapping classes 4. Named layers → immediate GradCAM and visualization hooks 5. Fully compatible with SageMaker Training jobs → just call build_model() inside your training script --- ## Meta - Lab notebook exported to Markdown for context and continuity. - Conversation lag resolved by starting a fresh thread with this summary as reference.

ChatGPT said:

Got it — thanks for dropping the lab notebook context. I've read through it carefully, and here's how I'll keep it in mind as we continue:

What You're Working On

- **Scaffolding scripts** (structure.ps1, structure.sh) to create project directories and placeholder files with tags.
- **Cross-platform focus:** CMD fallback exists, but the main emphasis is PowerShell (local dev, Windows) and Bash (AWS).
- **Testing:** Intend to confirm structure with tree or equivalent commands.
- **Next steps:** 30-min local push today (*Done-over-perfect* sprint), AWS work tomorrow, and vision draft Monday/Tuesday.

Key Considerations I'll Track

- ✅ Cross-platform: make sure solutions/scripts are portable between PowerShell and Bash.
- ✅ tree output: expected format important for validation.
- ✅ Execution quirks: e.g., ExecutionPolicy in PowerShell, file permissions (chmod) in Bash.
- ✅ Done-over-perfect approach: minimal viable scaffolding > polish later.
- ✅ Build model scaffolding (build_model.py): keep ready for AWS/ResNet-50++ experiments, but light footprint first.
- ✅ GradCAM/logging tasks: note them as future AWS priorities.

Immediate Questions You Raised

1. **PowerShell tree command** → I'll show you how to approximate it, since Windows doesn't have tree natively in PowerShell the way Linux has it in Bash.
2. **ExecutionPolicy concerns** → I'll give you the rundown on when powershell.exe -ExecutionPolicy RemoteSigned is needed (and when it's overkill).

Would you like me to address those two (PowerShell tree + ExecutionPolicy) now, so you've got them handy before your 30-minute sprint?

You said:

Please address the PowerShell stuff, then I'll eat and start my 30-minute timer.

ChatGPT said:

Here's a focused rundown on the **PowerShell pieces** you asked about, so you'll have it ready before you start your sprint:

---

1. Does PowerShell have a tree command?

- **Yes, but not natively inside PowerShell.**
  The tree utility is a classic Windows console command that comes with cmd.exe, not PowerShell itself.
- In PowerShell, you can still run it directly:
- tree /f /a
  - /f → list files as well as directories
  - /a → use ASCII characters (useful if Unicode tree characters don't render well in your terminal)
- Example:
- tree /f /a C:\path\to\project
- **Alternative inside PowerShell** (if you want something more native):
- Get-ChildItem -Recurse | Format-Table FullName

or for a tree-like view:

```
Get-ChildItem -Recurse | Sort-Object FullName | ForEach-Object {
  ($_ -replace '[^\\]+\\', '   ') + ($_ | Split-Path -Leaf)
}
```

(a bit hacky—good enough for checking structure, but tree is simpler and usually installed by default on Windows).

---

2. Do you need to use

```
powershell.exe -ExecutionPolicy RemoteSigned
```

- **What ExecutionPolicy is:** a PowerShell safety feature that controls whether scripts can run. By default, many Windows installs set it to Restricted or RemoteSigned.
- **Typical workflow:**
  - If you're writing your own local script (like structure.ps1) and saved it yourself, RemoteSigned allows it to run without nagging.
  - If you don't change anything, you may see:
  - File C:\...\structure.ps1 cannot be loaded because running scripts is disabled on this system.
- **One-time run option (safe for local testing):**
- powershell.exe -ExecutionPolicy Bypass -File .\structure.ps1
  - Bypass is less restrictive than RemoteSigned but temporary (affects only that run).
- **Permanent adjustment (user-level, not system-wide):**
- Set-ExecutionPolicy -Scope CurrentUser RemoteSigned

That way you won't have to specify it every time.

- **Bottom line:**
  - For *quick local sprints*, you can use the -ExecutionPolicy Bypass flag on just that run.
  - If you'll be scripting regularly, set RemoteSigned once for your user account and forget about it.

✅ **Recommendation for you today:**

- Just run tree /f /a $RootTest after your test run.
- If you hit an ExecutionPolicy block, either:
- Set-ExecutionPolicy -Scope CurrentUser RemoteSigned

(permanent, safe)
or

```
powershell.exe -ExecutionPolicy Bypass -File .\structure.ps1
```

(temporary).

Do you want me to also give you a **tiny wrapper function** for PowerShell that mimics tree with Get-ChildItem, so you don’t need to context-switch to CMD when you’re on Windows?

You said:

No, I don't want to review setting up a tree like alias. I think the

powershell

```
Get-ChildItem -Recurse | Sort-Object FullName | ForEach-Object {
  ($_ -replace '[^\\]+\\', '   ') + ($_ | Split-Path -Leaf)
}
```

should work fine. *Done-over-perfect*. ;)

ChatGPT said:

Perfect — that’s very much in the *Done-over-perfect* spirit.

The snippet with Get-ChildItem and ForEach-Object is absolutely fine for quick inspection. It won’t be as pretty as the ASCII tree, but it’ll give you the hierarchy and filenames inline without any setup. Good enough to confirm whether your scaffolding scripts did the right thing — then you can get on with the sprint.

So you’re set:

- Run your scaffolding script.
- Use tree /f /a $RootTest (classic way) **or** your PowerShell one-liner.
- If ExecutionPolicy nags you, bypass just for that run, or set it once and move on.

Eat, then sprint. 🚀 Would you like me to also give you a **ready-to-paste minimal test block** for PowerShell (tags + structure check) so you don’t have to retype before the 30-minute push?

You said:

Yes, that will be nice. I'm going with the scaffolding script, then I'll try tree /f /a $RootTest

ChatGPT said:

Here’s a minimal ready-to-paste block for PowerShell that fits your scaffolding test and then checks the structure with the classic tree command:

```
# Define root directory for test
$RootTest = "$PWD/test_project"

# Run your scaffolding script with two tags
.\structure.ps1 -RootDir $RootTest -Tags "p_01" "p_02"

# Check the resulting structure with the built-in Windows tree command
tree /f /a $RootTest
```

That’s all you need for your sprint:

1. Copy/paste into your PowerShell session.
2. Run.
3. Confirm directory + file names look right.

If you hit an ExecutionPolicy block, just prepend with:

```
powershell.exe -ExecutionPolicy Bypass -File .\structure.ps1 -RootDir $RootTest -Tags "p_01" "p_02"
```

…but likely you won't need it if you already allowed scripts on your machine.

This should let you eat, hit the 30-minute timer, and get immediate feedback with minimal friction. 🚀

You said:

Okay. I haven't done PowerShell in a while. I'll give you the first part of the error output I got, then I'll cat the output of structure.ps1, so we can debug.

powershell

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> $RootTest = "$PWD/test_project"                    PS C:\David\my_repos_dwb\fhtw-paper-code-prep> powershell -ExecutionPolicy Bypass -File .\structure.ps1 -RootDir $RootTest -Tags "p_01" "p_02"
At C:\David\my_repos_dwb\fhtw-paper-code-prep\test_project\scripts\ps_touch.ps1:1 char:17
+ param([string[]]\)
+                ~
Parameter declarations are a comma-separated list of variable names with optional initializer
expressions.
At C:\David\my_repos_dwb\fhtw-paper-code-prep\test_project\scripts\ps_touch.ps1:1 char:17
+ param([string[]]\)
+                ~
Missing ')' in function parameter list.
```

```
At C:\David\my_repos_dwb\fhtw-paper-code-prep\test_project\scripts\ps_touch.ps1:2 char:10
+ foreach (\ in \) {
+          ~
Missing variable name after foreach.
At C:\David\my_repos_dwb\fhtw-paper-code-prep\test_project\scripts\ps_touch.ps1:2 char:16
+ foreach (\ in \) {
+                ~
Unexpected token ')' in expression or statement.
    + CategoryInfo          : ParserError: (:) [], ParentContainsErrorRecordException
    + FullyQualifiedErrorId : InvalidFunctionParameter
```

That repeats quite a few times. There was a directory tree set up, and I'll send it if you need it. Here's the content of structure.ps1, output via cat.

powershell

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> cat .\structure.ps1
param(
    [string]$RootDir = "$PWD\new_experiment_dir",
    [string[]]$Tags = @("default")
)

# Ensure root directory exists
if (-not (Test-Path $RootDir)) { New-Item -ItemType Directory -Path $RootDir | Out-Null }

# Create a helper script for touching files
```

```
$TouchScript = Join-Path $RootDir "scripts\ps_touch.ps1"
if (-not (Test-Path $TouchScript)) {
    New-Item -ItemType Directory -Path (Split-Path $TouchScript) -Force | Out-Null
    @"
param([string[]]\$Paths)
foreach (\$p in \$Paths) {
    \$file = Resolve-Path -Path \$p -ErrorAction SilentlyContinue
    \$dir = Split-Path \$p
    if (-not (Test-Path \$dir)) { New-Item -ItemType Directory -Path \$dir | Out-Null }
    if (-not (Test-Path \$p)) { New-Item -ItemType File -Path \$p | Out-Null }
}
"@ | Set-Content -LiteralPath $TouchScript
}

# Files to create
$Files = @(
    "README.md",
    "notebooks\00_data_exploration.ipynb",
    "notebooks\01_model_build.ipynb",
    "notebooks\02_training.ipynb",
    "notebooks\03_inference_quick_explore.ipynb",
    "scripts\py_build_model.py",
    "scripts\build_model.ps1",
    "scripts\py_train_model.py",
    "scripts\train_model.ps1",
    "scripts\py_inference.py",
    "scripts\inference.ps1",
    "scripts\py_utils.py"
)

foreach ($tag in $Tags) {
    $TagDir = Join-Path $RootDir $tag

    # Create main directories
    $Dirs = @("notebooks", "datasets", "models", "logs", "scripts", "visualizations", "outputs\csv_logs", "outputs\gradcam_images")
    foreach ($d in $Dirs) { New-Item -ItemType Directory -Path (Join-Path $TagDir $d) -Force | Out-Null }

    # Create files with tag appended to stem
    foreach ($f in $Files) {
        $base = [System.IO.Path]::GetFileNameWithoutExtension($f)
        $ext  = [System.IO.Path]::GetExtension($f)
        $tagged = Join-Path $TagDir "$base`_$tag$ext"
        & powershell -File $TouchScript -Paths $tagged
    }
```

```
}


Write-Host "Project scaffolding with tags and subdirectories created at $RootDir"
PS C:\David\my_repos_dwb\fhtw-paper-code-prep>
```

ChatGPT said:

Thought for 16s

Great—thanks for the clean repro. The error is exactly what it looks like: the generated ps_touch.ps1 contains literal backslashes before the $ signs (e.g., \$Paths, \$p). That happened because the here-string you used was **double-quoted** (@" … "@) and you tried to “escape” the variables with backslashes. PowerShell doesn’t use \ to escape $ in strings, so those backslashes were written **into** the file and then the parser choked on \$….

Two solid fixes:

Fix A (best): use a single-quoted here-string to generate ps_touch.ps1

Single-quoted here-strings (@' … '@) **do not** expand variables, so you can write the script exactly as you want it to appear—no backslashes needed.

Replace your touch-script block with this:

```
# Create a helper script for touching files
$TouchScript = Join-Path $RootDir "scripts\ps_touch.ps1"
if (-not (Test-Path $TouchScript)) {
  New-Item -ItemType Directory -Path (Split-Path $TouchScript) -Force | Out-Null
  @'
param([string[]]$Paths)
foreach ($p in $Paths) {
  $dir = Split-Path -Parent $p
  if (-not (Test-Path $dir)) {
    New-Item -ItemType Directory -Path $dir -Force | Out-Null
  }
  if (-not (Test-Path $p)) {
    New-Item -ItemType File -Path $p -Force | Out-Null
  }
}
'@ | Set-Content -LiteralPath $TouchScript -Encoding UTF8
}
```

Notes:

- Single quotes on the here-string **prevent** expansion, so you can write $Paths and $p normally.
- I added -Parent to Split-Path (clearer, avoids accidental behavior changes).
- I added -Force where sensible and -Encoding UTF8 for consistency.
- Your later call can stay as-is, but if ExecutionPolicy allows, calling the script path directly is faster/cleaner:
- `& $TouchScript -Paths $tagged`

If you prefer to keep the separate process, do:

```
powershell -ExecutionPolicy Bypass -File $TouchScript -Paths $tagged
```

Fix B (simplest): don’t generate a helper script—do the touch inline

If you’re okay dropping the separate ps_touch.ps1, you can just replace the call with inline “touch” logic:

```
foreach ($f in $Files) {
  $base   = [System.IO.Path]::GetFileNameWithoutExtension($f)
  $ext    = [System.IO.Path]::GetExtension($f)
  $tagged = Join-Path $TagDir "$base`_$tag$ext"

  $dir = Split-Path -Parent $tagged
  if (-not (Test-Path $dir)) {
    New-Item -ItemType Directory -Path $dir -Force | Out-Null
  }
  if (-not (Test-Path $tagged)) {
    New-Item -ItemType File -Path $tagged -Force | Out-Null
  }
}
```

This avoids the whole quoting/expansion pitfall and is perfectly fine for *Done-over-perfect*.

Quick sanity checklist for your sprint

- Re-run with the **single-quoted here-string** (Fix A) **or** inline touch (Fix B).
- If you keep the helper, prefer calling it directly:
- & $TouchScript -Paths $tagged

(Use the powershell -ExecutionPolicy Bypass -File ... form only if policy blocks direct invocation.)

- Then your test block:
- $RootTest = "$PWD/test_project"
- .\structure.ps1 -RootDir $RootTest -Tags "p_01" "p_02"
- tree /f /a $RootTest

If anything else errors, paste the next few lines and we'll patch it fast so you can start that 30-minute timer.

You said:

All right. When I'm learning something new, I don't auto-allow scripts, because it keeps me in good practice to run the scripts on other machines at any company for which I might be working. In other words, I'm practicing when I have to use powershell -ExecutionPolicy Bypass -File .... I think that's a good thing to remember when an interview is coming up. (I'm also noting the Set-ExecutionPolicy -Scope CurrentUser RemoteSigned, but I'd initially start with the former command and note, "I don't want to change anything about your PowerShell without permission."

ChatGPT said:

That's a really sharp practice, and you're absolutely right — it shows both **technical caution** and **professional judgment**:

Why it's Good Practice

- **Security awareness:** Running with -ExecutionPolicy Bypass on a one-off basis demonstrates you know the security defaults exist for a reason and you're not cavalier about disabling them permanently.
- **Portability:** Many enterprise environments lock down user policies. By practicing this way, you're building muscle memory for how to run scripts safely on a locked-down workstation.
- **Interview advantage:** Being able to say *"I deliberately didn't change the machine's execution policy, because in a real workplace I wouldn't want to change someone else's security settings without permission"* is a solid plus. It shows you balance **getting the job done** with **respect for system integrity**.

When to Mention Set-ExecutionPolicy

- Good to mention you **know about** it (-Scope CurrentUser RemoteSigned is the least intrusive way).
- Even better to frame it as: *"If I owned the environment and wanted smoother daily scripting, I'd use RemoteSigned at the user scope. But when I'm a guest or contractor, I stick with Bypass on just that run."*
- That communicates flexibility: you're not unaware of easier ways, you're just being deliberate.

Extra Bonus Point

If asked in an interview: *“Why do you do that?”*, you can anchor your answer to security principles:

- **Principle of least privilege** → don’t change global/user defaults without a reason.
- **Reversibility** → one-time bypass is non-persistent, nothing to undo.
- **Awareness of defaults** → you understand why the system blocked the script in the first place.

👉 Practicing with -ExecutionPolicy Bypass instead of auto-allowing is the right call. It will both keep your habits sharp and give you a professional story that resonates in technical interviews.

Do you want me to also sketch a **short, interview-ready answer** (2–3 sentences) you could give if someone asks *“Why didn’t you just change the ExecutionPolicy?”*

You said:

Nope, what you gave has changed any answer I would give to a similar question from quite good to great ;) I'm going to try the new PowerShell structure.ps1. First, though, I need to remove the <path>\fhtw-paper-code-prep\test_project directory. I was thinking rmdir -Force test_project (my working directory is fhtw-paper-code-prep). That seems like it might be a mix of CMD and PowerShell. Is there a different PowerShell method (other than something like capitalizing RmDir?)

ChatGPT said:

You’re right to pause there — rmdir is actually an **alias in PowerShell**, but it behaves a bit differently than CMD’s rmdir.

PowerShell ways to remove a directory

- The canonical cmdlet is:
- Remove-Item -Recurse -Force .\test_project
    - -Recurse → removes all children
    - -Force → removes hidden/system items without prompting
- Aliases you can use interchangeably in PowerShell:

  - rm
  - rmdir
  - rd
  - del

So, in PowerShell:

```
rmdir -Recurse -Force .\test_project
```

works fine — the capitalization (RmDir, rmdir, RMDIR) doesn’t matter.

Quick guidance

- **Safer practice**: always add -Recurse -Force when removing directories, otherwise PowerShell will refuse to delete non-empty dirs.
- If you want a “dry run” before deleting, run:
- `Get-ChildItem .\test_project -Recurse`

to confirm what you’re about to wipe.

✅ So your instinct was right: in your working dir, just do:

```
rmdir -Recurse -Force .\test_project
```

Want me to also give you the **Bash equivalent** for when you’re testing the scaffolding script on Linux/AWS, so you don’t have to pause later?

You said:

I feel a lot better about bash, but I want to confirm. The dry run (list of files that would be deleted by rm -rf) is find ./test_project/ I could add a -print, but it shouldn't be necessary this time.

ChatGPT said:

You’ve got the right instinct 👍

In Bash/Linux:

- **Actual delete (no confirmation):**
- rm -rf ./test_project
- **Dry run (list contents first):**
- find ./test_project/

That will print every file/dir under test_project. Adding -print is redundant because find defaults to printing matches.

A couple of refinements

- If you want to **see exactly what would be deleted**, but not delete yet:
- find ./test_project/ -exec echo {} \;

(echo instead of rm shows the paths).

- For **directories only** (to preview structure):
- find ./test_project/ -type d

So yes, your version is already correct. Think of it as the Bash equivalent of PowerShell’s Get-ChildItem -Recurse before running Remove-Item.

Do you want me to also give you a one-liner alias-like trick, so you can type something like rmdry test_project and have it expand to that find dry run, or are you sticking with the explicit commands for now?

You said:

I'd love that alias for bash, especially the name, rmdry. I want to avoid the time necessary to remember how to work with the equivalent of ~/.bashrc in Powershell, so I'll just use the command for now. I do wonder if there's an equivalent to rm -i -rf in Powershell, in case I need to be careful in checking all the deletions, though.

ChatGPT said:

Got it — let’s tackle both pieces:

Bash alias for rmdry

You can add this to your ~/.bashrc (or ~/.bash_profile depending on your system):

```
alias rmdry='find "$1" -exec echo {} \;'
```

Usage:

```
rmdry ./test_project
```

That will print every file/dir that *would* be deleted by rm -rf. You can refine the alias later (e.g., make it check if $1 exists, or only list files with -type f).

PowerShell equivalent of rm -i -rf

PowerShell doesn't have an -i flag the way GNU rm does, but you can approximate it:

- **Prompt before each removal**:
- `Get-ChildItem -Recurse .\test_project | ForEach-Object {`
- `  $ans = Read-Host "Delete $($_.FullName)? (y/n)"`
- `  if ($ans -eq 'y') { Remove-Item $_.FullName -Force }`
- `}`
- **Safer, directory-level confirmation**:
- `Remove-Item -Recurse -Confirm .\test_project`

The -Confirm switch tells PowerShell to ask before it deletes the target(s).
If you do it on a big tree, it'll ask for the top-level item and then cascade, depending on your $ConfirmPreference.

- **Interactive only if uncertain**: You can also set $ConfirmPreference = 'High' before running a removal command to force more prompts.

✅ **Quick takeaway**

- **Linux:** rm -i -rf → prompts before each deletion.
- **PowerShell:** Remove-Item -Recurse -Confirm (or loop with Read-Host) is the closest match.

Would you like me to give you a **paired cheat-sheet block** (Bash vs. PowerShell) for deletion commands (dry run, force, interactive) so you can keep them side by side in your notebook?

You said:

I try to be filename-conscious, but I don't know if it's necessary in this case to defensively code for spaces, newlines, other weird things in filenames. Is alias rmdry='find "$1" -print0 | xargs -I'{}' -0 echo {} overkill, here?

ChatGPT said:

Thought for 10s

Great question. Short answer: your -print0 | xargs -0 approach is **not overkill** if you truly want to be resilient to *any* path weirdness (spaces, tabs, newlines, leading dashes, etc.). That said, there's an even cleaner option that avoids xargs entirely.

A few key points first:

- A Bash **alias** doesn't handle $1; you want a **shell function**.
- echo can mangle output (escapes, -n interpretation). Prefer printf.
- find … -exec … {} \; already passes a single filename *safely* as one argument—no need for -print0 | xargs -0 unless you specifically want a null-delimited pipeline for other reasons.

Here are two good, copy-pasteable options for your ~/.bashrc.

Option A (simple & safe, no xargs)

Handles spaces, tabs, newlines robustly by avoiding xargs and using printf.

```
rmdry() {
  if [ $# -ne 1 ]; then
    printf 'Usage: rmdry <path>\n' >&2
    return 2
  fi
  local target=$1
  # Show what would be removed (all entries under target), one per line
  find -- "$target" -mindepth 1 -exec printf '%s\n' {} \;
}
```

Notes:

- -- prevents weird “path looks like an option” issues (e.g., names starting with -).
- -mindepth 1 avoids printing the top directory itself (mirrors rm -rf target behavior).
- This works even if filenames contain **newlines** (they’ll still appear as multiple lines—there’s no truly newline-proof *human-readable* print format; if you need that, see Option B’s %q tip).

Option B (null-delimited pipeline, maximal robustness)

If you want to be fully explicit about delimiter safety:

```
rmdry() {
  if [ $# -ne 1 ]; then
    printf 'Usage: rmdry <path>\n' >&2
    return 2
  fi
  local target=$1
  find -- "$target" -mindepth 1 -print0 \
    | xargs -0 -n1 -I{} printf '%s\n' "{}"
}
```

This is very resilient and nice when you later swap printf for another tool that *requires* null-delimited input.

Optional: show *escaped* paths (easy to copy/paste)

If you'd like to see shell-escaped paths (so you can copy a line and paste it safely):

```
rmdryq() {
  if [ $# -ne 1 ]; then
    printf 'Usage: rmdryq <path>\n' >&2
    return 2
  fi
  local target=$1
  # Bash-specific %q to print a representation you can safely reuse in a shell
  while IFS= read -r -d '' p; do
    printf '%q\n' "$p"
  done < <(find -- "$target" -mindepth 1 -print0)
}
```

Which should you use?

- Want the simplest, portable solution? **Option A**.
- Want strict delimiter safety / future-proof piping into other tools? **Option B**.

- Want copy/paste-safe escaping? Use **rmdryq** occasionally.

All three are “filename-conscious” and safe for the real world.

You said:

Awesome, I'd like that paired cheat-sheet block. I think I might have strayed a bit from the *Done-over-perfect*, but I feel better knowing these things for PowerShell. I'm pretty sure Amazon AWS will usually prefer Linux, but you never know.

ChatGPT said:

Here’s the **paired cheat-sheet block** you asked for — Bash vs. PowerShell deletion patterns side-by-side. This way you’ve got a single reference for *force*, *interactive*, and *dry run* across both environments.

🗑 Deletion Cheat-Sheet: Bash vs. PowerShell

Purpose	Bash / Linux	PowerShell
Force remove non-empty dir (no prompt)	rm -rf ./test_project	Remove-Item -Recurse -Force .\test_project Aliases: rm, rmdir, rd, del
Interactive remove (confirm before each)	rm -i -rf ./test_project	Remove-Item -Recurse -Confirm .\test_project OR loop with: Get-ChildItem -Recurse .\test_project | ForEach-Object { $ans = Read-Host "Delete $($_.FullName)? (y/n)"; if ($ans -eq 'y') { Remove-Item $_.FullName -Force } }
Dry run (list what would be deleted)	find ./test_project/ (default lists all) Safer: find ./test_project/ -exec echo {} \; Robust alias/function: rmdry as discussed	Get-ChildItem -Recurse .\test_project Quick preview of all files/dirs before deletion

---

✅ Quick notes

- **Bash**: find is the easiest “dry run.” For strict filename safety, your rmdry function with -print0 + xargs -0 is excellent.
- **PowerShell**: Remove-Item -Recurse -Confirm is the closest analog to rm -i -rf. If you need per-file prompts, wrap with a Read-Host loop.
- **Cross-platform takeaway**: On AWS/Linux you’ll almost always use Bash, but practicing the PowerShell forms is excellent for interviews and for Windows dev.

Would you like me to also prepare a **tiny “portable checklist” snippet** (just the commands without explanations, in two neat blocks) that you can paste into your lab notebook for sprint sessions, so you don’t have to wade through notes each time?

You said:

Perfect. I have a template of such short-snippet versions of oft-used code, and such a version for this is welcome. (One of my most up-voted answers on the StackExchange network shows an empirical comparison of Linux and CMD rm, rmdir, del, etc. on a simple directory structure. Now I have an extension of it.

ChatGPT said:

Nice — here’s the **short-snippet portable checklist** version you can drop straight into your template. It mirrors the longer cheat-sheet but stripped down to bare commands for quick use during sprints.

🔷 Bash / Linux

```
# Force remove

rm -rf ./test_project



# Interactive remove

rm -i -rf ./test_project
```

```
# Dry run
find ./test_project/
# (or robust)
find ./test_project/ -exec echo {} \;

# With alias/function
rmdry ./test_project
```

🔷 **PowerShell**

```
# Force remove
Remove-Item -Recurse -Force .\test_project
# aliases: rm, rmdir, rd, del

# Interactive remove (confirm)
Remove-Item -Recurse -Confirm .\test_project

# Per-file prompt (optional)
Get-ChildItem -Recurse .\test_project | ForEach-Object {
  $ans = Read-Host "Delete $($_.FullName)? (y/n)"
  if ($ans -eq 'y') { Remove-Item $_.FullName -Force }
}
```

```
# Dry run (list only)

Get-ChildItem -Recurse .\test_project
```

This way you've got the **full range of remove behaviors** (force / interactive / dry run) in a glanceable form. It's exactly the kind of snippet that's perfect to keep alongside your earlier SE comparison work.

Would you like me to also prep the **PowerShell + Bash alias/function definitions** (like your rmdry) in a minimal form, so you can just paste them into ~/.bashrc or keep the PS loop handy without notes?

You said:

Yes, then I'll run Remove-Item -Recurse -Force .\test_project. I already saw all the contents with the tree /a /f, so I'm not to concerned. Then I'll try the new structure.ps.

ChatGPT said:

Great — here are the **minimal alias/function definitions** for Bash and PowerShell so you can keep them handy. These are stripped down for quick copy-paste into ~/.bashrc (Linux) or just inline use in PowerShell.

🔷 Bash (~/.bashrc or inline)

```
# Dry-run rm (list what would be deleted)

rmdry() {

  if [ $# -ne 1 ]; then

    printf 'Usage: rmdry <path>\n' >&2

    return 2

  fi

  find -- "$1" -mindepth 1 -exec printf '%s\n' {} \;
}
```

Usage:

```
rmdry ./test_project
```

🔷 PowerShell (inline use — paste directly in a session)

```
function RmDry {
  param([string]$Path)
  if (-not (Test-Path $Path)) {
    Write-Error "Path not found: $Path"
    return
  }
  Get-ChildItem -Recurse $Path
}
```

Usage:

```
RmDry .\test_project
```

✅ With these defined:

- Bash → rmdry ./dir
- PowerShell → RmDry .\dir

Now you can go ahead and safely run:

```
Remove-Item -Recurse -Force .\test_project
```

…then test your new structure.ps1.

Do you want me to also prep a **success-criteria checklist** for your structure.ps1 run (what you should see in tree output to confirm the fix worked)? That way you'll know in <30 seconds if you're good to move on.

You said:

Can I have you serve as a @TODO tag for if there's extra time after *Done-not-perfect*? I'd like to review aliases (and functions and settings; basically the ~/.bashrc/ equivalent) for PowerShell, but that would be more "Perfect" than "Done" right now.

ChatGPT said:

Absolutely — I can act as your **@TODO tag** for "review PowerShell aliases/functions/settings (the ~/.bashrc equivalent)" later. That way you can stay in *Done-not-perfect* mode now, and circle back only if there's extra time or after the sprint.

I'll keep it in mind as a deferred task. ✅

So immediate flow for you now:

1. Remove-Item -Recurse -Force .\test_project
2. Re-run structure.ps1 with your tags.
3. Quick check with tree /a /f $RootTest (or Get-ChildItem -Recurse).

If the directory layout looks right, sprint complete. 🚀

Want me to also jot down a **one-liner success criteria** for the tree output (so you can glance and know "yep, fixed")?

You said:

Yeah, that prevents me from having to write a test.

ChatGPT said:

Here's a simple **success-criteria one-liner** for your structure.ps1 sprint — no need to write a formal test, just eyeball the output:

✅ Success criteria for tree /a /f $RootTest

- For each tag (e.g. p_01, p_02), you should see:
    - Top-level subdir named after the tag
    - Inside it: the expected folders (notebooks, datasets, models, logs, scripts, visualizations, outputs\csv_logs, outputs\gradcam_images)
    - In those folders: files with the tag appended, e.g.
    - README_p_01.md
    - 00_data_exploration_p_01.ipynb
    - py_build_model_p_01.py
    - train_model_p_01.ps1
- Repeat for p_02 with _p_02 suffixes.

👉 If you can scroll the tree output and spot **two sets of correctly suffixed files and directories**, the scaffolding script did its job. No need to check line-by-line during the sprint.

Want me to also give you a **sample expected tree snippet** (abbreviated, not the full thing) so you have a mental picture to compare against when it scrolls by?

You said:

All right. I think I'm good for lunch.

powershell

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> Remove-Item -Recurse -Force .\test_project\

PS C:\David\my_repos_dwb\fhtw-paper-code-prep> # Note that I ran into issues with the Get-ChildItem version that weren't worth debugging}
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> tree /a /f
## I'll give the part that had changed, below ##
```

New part of tree output.

powershell

```
|
\---test_project
    \---p_01
        |   00_data_exploration_p_01.ipynb
        |   01_model_build_p_01.ipynb
        |   02_training_p_01.ipynb
        |   03_inference_quick_explore_p_01.ipynb
        |   build_model_p_01.ps1
        |   inference_p_01.ps1
        |   py_build_model_p_01.py
        |   py_inference_p_01.py
        |   py_train_model_p_01.py
        |   py_utils_p_01.py
        |   README_p_01.md
        |   train_model_p_01.ps1
        |
        +---datasets
        +---logs
        +---models
        +---notebooks
        +---outputs
        |   +---csv_logs
        |   \---gradcam_images
        +---scripts
        \---visualizations
```

That's exactly what I was expecting. Hooray!

ChatGPT said:

🎉 That’s a win! The tree output matches the success criteria exactly:

- **Tag directory (p_01) created**
- **All expected notebooks, scripts, and README tagged correctly**
- **Subdirectories (datasets, logs, models, notebooks, outputs, scripts, visualizations) scaffolded properly**
- **Nested outputs (csv_logs, gradcam_images) are in place**

That means your structure.ps1 fix worked as intended. Nice job sticking with the *Done-over-perfect* principle — you debugged only what mattered, got the expected tree, and you’re ready to move forward. 🚀

Enjoy your lunch — when you come back, you can start the 30-minute sprint knowing the scaffolding is solid. Do you want me to keep a short “next immediate step” reminder handy for when you return, so you don’t have to scroll back through?

You said:

Let me add in the version of structure.ps1 that gave the desired result. I'm much more likely to search for it in a Lab Notebook than in a local-test directory (which I might have deleted by the time I need it).

powershell

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> cat .\structure.ps1
param(
  [string]$RootDir = "$PWD\new_experiment_dir",
  [string[]]$Tags = @("default")
)

# Ensure root directory exists
if (-not (Test-Path $RootDir)) { New-Item -ItemType Directory -Path $RootDir | Out-Null }

###  No helper required; this was just paralleling our CMD pattern
###+ set up to avoid the obstruse `$ >> filename 2>nul`
## Create a helper script for touching files
##$TouchScript = Join-Path $RootDir "scripts\ps_touch.ps1"
##if (-not (Test-Path $TouchScript)) {
##    New-Item -ItemType Directory -Path (Split-Path $TouchScript) -Force | Out-Null
##    @'
##param([string[]]\$Paths)
##foreach (\$p in \$Paths) {
##    \$file = Resolve-Path -Path \$p -ErrorAction SilentlyContinue
##    \$dir = Split-Path \$p
##    if (-not (Test-Path \$dir)) { New-Item -ItemType Directory -Path \$dir | Out-Null }
##    if (-not (Test-Path \$p)) { New-Item -ItemType File -Path \$p | Out-Null }
##}
##'@ | Set-Content -LiteralPath $TouchScript
##}

# Files to create
$Files = @(
  "README.md",
  "notebooks\00_data_exploration.ipynb",
  "notebooks\01_model_build.ipynb",
  "notebooks\02_training.ipynb",
  "notebooks\03_inference_quick_explore.ipynb",
  "scripts\py_build_model.py",
  "scripts\build_model.ps1",
  "scripts\py_train_model.py",
  "scripts\train_model.ps1",
  "scripts\py_inference.py",
  "scripts\inference.ps1",
  "scripts\py_utils.py"
)

foreach ($tag in $Tags) {
  $TagDir = Join-Path $RootDir $tag
  # Create main directories
  $Dirs = @("notebooks", "datasets", "models", "logs", "scripts", "visualizations", "outputs\csv_logs", "outputs\gradcam_images")
  foreach ($d in $Dirs) { New-Item -ItemType Directory -Path (Join-Path $TagDir $d) -Force | Out-Null }

  # Create files with tag appended to stem
  foreach ($f in $Files) {
    $base   = [System.IO.Path]::GetFileNameWithoutExtension($f)
    $ext    = [System.IO.Path]::GetExtension($f)
    $tagged = Join-Path $TagDir "$base`_$tag$ext"

    $dir = Split-Path -Parent $tagged
    if (-not (Test-Path $dir)) {
      New-Item -ItemType Directory -Path $dir -Force | Out-Null
    }
    if (-not (Test-Path $tagged)) {
      New-Item -ItemType File -Path $tagged -Force | Out-Null
    }
  }
}

Write-Host "Project scaffolding with tags and subdirectories created at $RootDir"
```

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep>
```

ChatGPT said:

Perfect — this final version of structure.ps1 is exactly the one that produced the correct scaffolding output. Recording it in your **Lab Notebook** is the right move: far more durable than stashing it in a throwaway test directory.

Here's what you now have captured:

✅ Key differences in the working version

- **No external helper script** (ps_touch.ps1) → simplified everything and avoided quoting pitfalls.
- **Inline file creation** inside the main loop (New-Item -ItemType File …) → direct, no subprocess needed.
- **Forced directory creation (-Force)** → makes reruns idempotent and cleaner.
- **Commented-out helper section** remains as documentation, showing the earlier CMD-style pattern you were paralleling, but safely ignored.

Why this is a solid baseline

- Easier to maintain across environments.
- Directly parallels the Bash scaffolding you'll likely use on AWS.
- Idempotent: re-running the script won't cause errors, just silently ensures the structure exists.
- Debug-friendly: every step is inline, no nested script to parse.

📌 I'd mark this one in your notes as something like:

structure.ps1 — inline "touch" version — tested good (2025-08-30).

That way you'll immediately know it's the **known-working revision**.

Do you want me to also give you a **tiny side-by-side snippet** showing the working inline PowerShell file-touch vs. the Bash equivalent, so you'll have a "cross-platform mental map" in one place for when you switch contexts?

You said:

Yep. I'm going to make that note in the pull request I'll start with all the changes.

ChatGPT said:

Here's a compact **cross-platform mental map** for your scaffolding scripts. This way you can keep the PowerShell and Bash equivalents side by side in your notes and quickly recall how each handles "touch" logic inline.

🔷 PowerShell (working inline version)

```
# Create tagged file safely (idempotent)

$dir = Split-Path -Parent $tagged

if (-not (Test-Path $dir)) {

  New-Item -ItemType Directory -Path $dir -Force | Out-Null

}

if (-not (Test-Path $tagged)) {

  New-Item -ItemType File -Path $tagged -Force | Out-Null

}
```

- **Split-Path -Parent** → get directory
- **New-Item -ItemType Directory -Force** → ensure dirs exist
- **New-Item -ItemType File -Force** → "touch"-like behavior

🔷 Bash (AWS-ready equivalent)

```
# Create tagged file safely (idempotent)

dir=$(dirname "$tagged")

[ -d "$dir" ] || mkdir -p "$dir"

[ -f "$tagged" ] || touch "$tagged"
```

- dirname → get directory
- mkdir -p → ensure dirs exist
- touch → creates file if missing

✅ Takeaway

- Both versions do:
    1. Get the parent directory path
    2. Create directory if missing
    3. Create file if missing
- **Idempotent** → re-running is safe, no duplicates, no errors.
- **Minimal differences**: PowerShell uses New-Item, Bash uses mkdir+touch.

This gives you a quick “Rosetta Stone” for the scaffolding logic across the two shells. Great to include in your pull request note so reviewers (and your future self) see the direct mapping.

Want me to also prepare a **mini-comment block** (two or three sentences) you could paste right above this code in your repo, explaining why the inline approach is preferred over the earlier helper-script version?

You said:

I'm going to use a tmp directory to ensure correct functioning of the bash script when I get back. I can debug bash pretty quickly, but I'll still set myself a ten-minute limit.

ChatGPT said:

That sounds like a really good plan 👍

- Using a tmp directory keeps your main repo clean and avoids accidental overwrite while debugging.
- Giving yourself a **ten-minute cap** fits perfectly with your *Done-not-perfect* principle — you'll either fix it quickly or move on without getting bogged down.
- Bash debugging is usually fast (errors tend to be obvious and fixable with echo or set -x), so ten minutes is realistic.

When you're back and testing the Bash script, you can do something like:

```
TMPDIR="$PWD/tmp_test_project"

./structure.sh "$TMPDIR" p_01 p_02

tree -a -F "$TMPDIR"
```

That way the whole test lives under tmp_test_project, and you can just rm -rf it when you're done.

Would you like me to prep a **minimal Bash debugging checklist** (like 3–4 quick commands: enable tracing, inspect variables, rerun) so you don't have to think about debugging steps during the sprint?

You said:

Yes, and then give something like a "here's what to start with when you get back." The stuff you've done as far as that goes is a much easier way to keep my momentum going despite ADHD than the write-it-on-a-laminated-sheet-with-a-dry-erase-marker stuff I've been doing.

ChatGPT said:

Got it — here's a **tight debugging checklist** plus a short **"when you get back" starter"** so you can dive in without thinking twice.

🔧 Minimal Bash Debugging Checklist (10-minute cap)

1. **Enable tracing**
2. set -x

(prints each command as it runs — turn off with set +x)

3. **Echo variables** at critical points in structure.sh
4. echo "ROOT_DIR=$ROOT_DIR"
5. echo "tagged=$tagged"
6. **Dry-run directory listing**
7. tree -a -F "$ROOT_DIR"

(confirms what was created without needing extra logic)

8. **Quick cleanup**
9. rm -rf "$ROOT_DIR"

(reset and rerun if you need to test again)

👉 If it’s not fixed within **10 minutes**, log the error, stop, and move on.

📌 When You Get Back

1. Create a temp dir and run your scaffolding script:
2. TMPDIR="$PWD/tmp_test_project"
3. ./structure.sh "$TMPDIR" p_01 p_02
4. tree -a -F "$TMPDIR"
5. If errors show up, enable tracing:
6. set -x

7. ./structure.sh "$TMPDIR" p_01 p_02
8. set +x
9. Fix the **first obvious issue only** (don’t chase every small one).
   If it works, great. If not in 10 minutes, jot down the failure mode and move on.

This way you’ll have a **launchpad** ready for post-lunch: minimal thinking required, just paste and go.

Want me to also add a **tiny “success picture”** (an abbreviated tree output snippet for Bash like you did for PowerShell) so you can immediately know if the script succeeded without scanning everything?

You said:

That's good. Note that the 30-minute sprint should include an attempt at the getting the following running correctly (build_model() pattern, which seems best-practice for both TF2 and PyTorch workflows) with CIFAR10's test/train split. This should go in a Jupyter notebook (created via Jupyter Lab)

python

```
import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses
import numpy as np
import matplotlib.pyplot as plt
import timeit

# For GradCAM
import cv2

# Optional: humanfriendly for readable time
```

```
import humanfriendly
python
from tensorflow.keras import layers, Model


def build_model(input_shape=(32,32,3),
                n_classes=10,
                filters=[32,64,64],
                fc_units=64,
                use_conv2=True,
                use_conv3=True,
                multi_label=False):
    """
    Builds a CNN similar to A-0 model with options for ablation and multi-label outputs.


    Args:
        input_shape (tuple): Input image shape, e.g., (32,32,3)
        n_classes (int): Number of output classes
        filters (list of int): Number of filters for conv1, conv2, conv3
        fc_units (int): Units in fully-connected layer
        use_conv2 (bool): Include conv2 block
        use_conv3 (bool): Include conv3 block
        multi_label (bool): True for multi-label classification, False for single-label
```

```
Returns:
    keras.Model: Compiled CNN model (logits if multi_label=False)
"""

inputs = layers.Input(shape=input_shape)

# --- Conv1 block ---
x = layers.Conv2D(filters[0], (3,3), padding='same', activation='relu', name='conv1')(inputs)
x = layers.MaxPooling2D((2,2))(x)

# --- Optional Conv2 block ---
if use_conv2:
    x = layers.Conv2D(filters[1], (3,3), padding='same', activation='relu', name='conv2')(x)
    x = layers.MaxPooling2D((2,2))(x)

# --- Optional Conv3 block ---
if use_conv3:
    x = layers.Conv2D(filters[2], (3,3), padding='same', activation='relu', name='conv3')(x)
    x = layers.MaxPooling2D((2,2))(x)

# --- Fully connected ---
```

```
  x = layers.Flatten()(x)

  x = layers.Dense(fc_units, activation='relu')(x)



  # --- Output layer ---

  if multi_label:

    outputs = layers.Dense(n_classes, activation='sigmoid', name='output')(x)

  else:

    outputs = layers.Dense(n_classes, name='output')(x)  # logits; use softmax externally if desired



  model = Model(inputs=inputs, outputs=outputs, name="A0_CNN")



  return model
```

We also drafted code for - Train Model - Temperature Scaling (Calibration) - Evaluate & Inspect Misclassifications - GradCAM Integration (simplified) but the *Done-not-perfect* idea leads me to set the goal of running the training locally and doing standard accuracy/loss plots. With evaluation on the test set coming if we don't go over 30 minutes. Any extra time can then be used on any other part.

ChatGPT said:

Thought for 9s

Love it. Here’s a **tight 30-minute sprint plan** plus **ready-to-paste notebook cells** that will (a) load CIFAR-10, (b) build your model, (c) train, (d) plot accuracy/loss, and (e) evaluate on the test set. Everything is set up for **logits + SparseCategoricalCrossentropy(from_logits=True)**, minimal friction, and speed. I’ve used **two-space indentation** and **heredoc-style docstrings** in all defs, per your preference.

---

0) (Optional) Repro & GPU sanity

```
"""
```

```
Set seeds for light reproducibility and print GPU info.
"""

import os, random, numpy as np, tensorflow as tf

def set_repro(seeds=(1337,)):
  """
  Set lightweight reproducibility seeds for Python, NumPy, and TensorFlow.
  Not bit-for-bit deterministic across all ops, but reduces variance.
  """
  seed = seeds[0] if seeds else 1337
  os.environ["PYTHONHASHSEED"] = str(seed)
  random.seed(seed)
  np.random.seed(seed)
  tf.random.set_seed(seed)

set_repro()

gpus = tf.config.list_physical_devices('GPU')
print("GPUs:", gpus if gpus else "None detected (CPU only)")
```

---

1) Imports (your list, plus a couple helpers)

```
"""
Core imports for CIFAR-10 training sprint.
"""

import tensorflow as tf
from tensorflow.keras import layers, models, optimizers, losses, callbacks
import numpy as np
import matplotlib.pyplot as plt
import time
# Optional: humanfriendly for readable time
try:
  import humanfriendly
except Exception:
  humanfriendly = None
```

---

2) Data: CIFAR-10 load + simple preprocessing

```
"""
Load CIFAR-10 and prepare tf.data pipelines.
"""

def get_cifar10(batch_size=128, shuffle_buffer=10000):
  """
```

```
    Load CIFAR-10, normalize to [0,1], and return batched train/val/test datasets.
    Uses a 45k/5k train/val split from the original 50k training set.
    """
    (x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()
    y_train = y_train.squeeze().astype(np.int64)
    y_test  = y_test.squeeze().astype(np.int64)

    # basic normalization to [0,1]
    x_train = x_train.astype(np.float32) / 255.0
    x_test  = x_test.astype(np.float32) / 255.0

    # train/val split
    x_tr, x_val = x_train[:45000], x_train[45000:]
    y_tr, y_val = y_train[:45000], y_train[45000:]

    ds_tr  = tf.data.Dataset.from_tensor_slices((x_tr, y_tr)).shuffle(shuffle_buffer).batch(batch_size).prefetch(tf.data.AUTOTUNE)
    ds_val = tf.data.Dataset.from_tensor_slices((x_val, y_val)).batch(batch_size).prefetch(tf.data.AUTOTUNE)
    ds_te  = tf.data.Dataset.from_tensor_slices((x_test, y_test)).batch(batch_size).prefetch(tf.data.AUTOTUNE)

    return ds_tr, ds_val, ds_te

ds_tr, ds_val, ds_te = get_cifar10()
```

---

3) Your build_model (as provided), unchanged

```
"""
Model builder (returns uncompiled model with logits if multi_label=False).
"""

from tensorflow.keras import layers, Model

def build_model(input_shape=(32,32,3),
                n_classes=10,
                filters=[32,64,64],
                fc_units=64,
                use_conv2=True,
                use_conv3=True,
                multi_label=False):
  """
  Builds a CNN similar to A-0 model with options for ablation and multi-label outputs.

  Args:
    input_shape (tuple): Input image shape, e.g., (32,32,3)
    n_classes (int): Number of output classes
    filters (list of int): Number of filters for conv1, conv2, conv3
    fc_units (int): Units in fully-connected layer
    use_conv2 (bool): Include conv2 block
    use_conv3 (bool): Include conv3 block
    multi_label (bool): True for multi-label classification, False for single-label

  Returns:
    keras.Model: Uncompiled CNN model (logits if multi_label=False)
  """
  inputs = layers.Input(shape=input_shape)

  # --- Conv1 block ---
  x = layers.Conv2D(filters[0], (3,3), padding='same', activation='relu', name='conv1')(inputs)
  x = layers.MaxPooling2D((2,2))(x)

  # --- Optional Conv2 block ---
  if use_conv2:
    x = layers.Conv2D(filters[1], (3,3), padding='same', activation='relu', name='conv2')(x)
    x = layers.MaxPooling2D((2,2))(x)

  # --- Optional Conv3 block ---
  if use_conv3:
    x = layers.Conv2D(filters[2], (3,3), padding='same', activation='relu', name='conv3')(x)
```

```
    x = layers.MaxPooling2D((2,2))(x)

  # --- Fully connected ---
  x = layers.Flatten()(x)
  x = layers.Dense(fc_units, activation='relu')(x)

  # --- Output layer ---
  if multi_label:
    outputs = layers.Dense(n_classes, activation='sigmoid', name='output')(x)
  else:
    outputs = layers.Dense(n_classes, name='output')(x)  # logits

  model = Model(inputs=inputs, outputs=outputs, name="A0_CNN")
  return model
```

---

4) Compile, train, time it

```
"""
Compile the model and run a short training with early stopping.
"""

def compile_model(model, lr=1e-3, multi_label=False):
  """
```

```
    Compile a Keras model for single-label (logits) or multi-label classification.
    """
    if multi_label:
        loss = tf.keras.losses.BinaryCrossentropy()
        metrics = [tf.keras.metrics.BinaryAccuracy(name="bin_acc")]
    else:
        loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
        metrics = [tf.keras.metrics.SparseCategoricalAccuracy(name="acc")]
    model.compile(
        optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
        loss=loss,
        metrics=metrics
    )
    return model


def train_quick(ds_tr, ds_val, epochs=8):
    """
    Train with EarlyStopping and return (history, seconds_elapsed).
    """
    es = callbacks.EarlyStopping(monitor="val_acc" if "acc" in ds_val.element_spec[1].dtype.name else "val_loss",
                                 patience=3, restore_best_weights=True)
    start = time.time()
```

```
  hist = model.fit(ds_tr, validation_data=ds_val, epochs=epochs, callbacks=[es], verbose=1)
  secs = time.time() - start
  return hist, secs


# build + compile
model = build_model()
model = compile_model(model, lr=1e-3, multi_label=False)


history, secs = train_quick(ds_tr, ds_val, epochs=8)
if humanfriendly:
  print("Train time:", humanfriendly.format_timespan(secs))
else:
  print(f"Train time: {secs:.1f}s")
```

5) Plot accuracy & loss

```
"""
Plot training curves for quick visual confirmation.
"""


def plot_history(history):
  """
  Plot accuracy and loss curves from a Keras History object.
  """
  h = history.history
  # Try common keys defensively
  acc_key = "acc" if "acc" in h else "sparse_categorical_accuracy" if "sparse_categorical_accuracy" in h else None
  val_acc_key = "val_acc" if "val_acc" in h else "val_sparse_categorical_accuracy" if "val_sparse_categorical_accuracy" in h else None

  plt.figure()
  if acc_key and val_acc_key:
    plt.plot(h[acc_key], label="train acc")
    plt.plot(h[val_acc_key], label="val acc")
    plt.xlabel("epoch"); plt.ylabel("accuracy"); plt.legend(); plt.title("Accuracy")
    plt.show()

  plt.figure()
  plt.plot(h["loss"], label="train loss")
  if "val_loss" in h:
    plt.plot(h["val_loss"], label="val loss")
  plt.xlabel("epoch"); plt.ylabel("loss"); plt.legend(); plt.title("Loss")
  plt.show()

plot_history(history)
```

---

6) Evaluate on the test set (keep it simple)

```
"""
Evaluate the trained model on the CIFAR-10 test set.
"""


def evaluate_model(model, ds_te):
  """
  Evaluate on test set and return a dict with loss and accuracy.
  """
  results = model.evaluate(ds_te, verbose=0, return_dict=True)
  print("Test results:", results)
  return results


_ = evaluate_model(model, ds_te)
```

Stretch goals (only if time remains)

- **Save artifacts:** model.save("models/a0_cnn.keras") and dump history.history to CSV.
- **Confusion matrix:** do a quick tf.math.confusion_matrix with argmax(logits).
- **GradCAM stub:** leave a commented cell header to plug in later (you listed it as very important).

Quick save helpers

```
"""
Optional: save model and training history.
```

```
"""

import json, os

def save_artifacts(model, history, out_dir="outputs"):
  """
  Save Keras model and training curves history to disk.
  """
  os.makedirs(out_dir, exist_ok=True)
  model.save(os.path.join(out_dir, "a0_cnn.keras"))
  with open(os.path.join(out_dir, "history.json"), "w") as f:
    json.dump(history.history, f)
  print(f"Saved model and history under {out_dir}/")

# Uncomment to save:
# save_artifacts(model, history)
```

What to aim for in the 30-minute sprint

- You should be able to reach **>50% val/test acc** within a handful of epochs with this tiny CNN (it won’t match ResNet-50++, but it’s enough to prove the pipeline).
- If training is slow on CPU, drop batch size to 64 or epochs to 4; the goal is a **clean run + plots**, not SOTA.
- If you want to squeeze a bit more, add a tiny preprocessing stack:

- aug = tf.keras.Sequential([
-   layers.RandomFlip("horizontal"),
-   layers.RandomRotation(0.05),
-])

and apply it in the dataset map, but only if you have time.

If you want next, I can hand you a **single cell** that adds **temperature scaling** and prints ECE (just a minimal wrapper using validation logits) — but I'd keep that as a post-sprint task.

You said:

Thanks, by the way, for incorporating my style of two spaces rather than four (or a tab). Could you also use 137 as the repro seed? I like it or -137, what with that being the approximate inverse of the fine structure constant&mdash;the coupling strength for QED. Got to get those callbacks to physics in ;)

ChatGPT said:

That's a fantastic callback — I love the physics tie-in. Let's swap in **137** as your reproducibility seed, keeping the two-space indentation and heredoc-style docstrings.

Here's the revised **seed setup cell**:

```
"""
Set seeds for light reproducibility and print GPU info.
"""


import os, random, numpy as np, tensorflow as tf


def set_repro(seed=137):
  """
  Set lightweight reproducibility seeds for Python, NumPy, and TensorFlow.
  Not bit-for-bit deterministic across all ops, but reduces variance.

  Args:
    seed (int): Preferred reproducibility seed (default 137).
  """
  os.environ["PYTHONHASHSEED"] = str(seed)
  random.seed(seed)
  np.random.seed(seed)
  tf.random.set_seed(seed)

set_repro(137)

gpus = tf.config.list_physical_devices('GPU')
print("GPUs:", gpus if gpus else "None detected (CPU only)")
```

Why 137 (or -137)?

- As you said: it's the **inverse of the fine structure constant α**, ~1/137 — the dimensionless coupling strength for QED.
- Perfect way to sneak physics callbacks into your ML workflow.
- Works symbolically, too: you can tell interviewers *"I seed with 137 because I like my experiments to couple as strongly to reality as QED does to electromagnetism."* 😄

Would you like me to also fold this set_repro(137) call directly into the **training cell** I gave earlier, so the seed is always set right before training? That way you don’t risk forgetting it during a sprint.

You said:

Yes, please do so. Also, after putting that code for the training cell, can you put the bash version (possibly after the PowerShell version) for the directory structure. If you can tweak it to have the tree output match the following, that will save some time as we go to AWS:

plain

```
test_project/
 ├– p_01/
|   ├– README_p_01.md
|   ├– notebooks/
|  |   ├– 00_data_exploration_p_01.ipynb
|  |   ├– 01_model_build_p_01.ipynb
|  |   ├– 02_training_p_01.ipynb
|  |  └– 03_inference_quick_explore_p_01.ipynb
|   ├– scripts/
|  |   ├– py_build_model_p_01.py
|  |   ├– build_model_p_01.cmd
|  |   ├– py_train_model_p_01.py
|  |   ├– train_model_p_01.cmd
|  |   ├– py_inference_p_01.py
|  |   ├– inference_p_01.cmd
|  |   ├– py_utils_p_01.py
|  |  └─ py_touch.py
|   ├─ datasets/
|   ├─ models/
|   ├─ logs/
|   ├─ visualizations/
|  └─ outputs/
|      ├─ csv_logs/
|     └─ gradcam_images/
 ├─ p_02/
|   ├─ README_p_02.md
|   ├─ notebooks/
|  |   ├─ 00_data_exploration_p_02.ipynb
|  |   ├─ 01_model_build_p_02.ipynb
|  |   ├─ 02_training_p_02.ipynb
|  |  └─ 03_inference_quick_explore_p_02.ipynb
|   ├─ scripts/
|  |   ├─ py_build_model_p_02.py
|  |   ├─ build_model_p_02.cmd
|  |   ├─ py_train_model_p_02.py
|  |   ├─ train_model_p_02.cmd
|  |   ├─ py_inference_p_02.py
| |   ├─ inference_p_02.cmd
| |   ├─ py_utils_p_02.py
| |  └─ py_touch.py
|   ├─ datasets/
|   ├─ models/
|   ├─ logs/
|   ├─ visualizations/
|  └─ outputs/
|      ├─ csv_logs/
|     └─ gradcam_images/
```

ChatGPT said:

Thought for 14s

Love the physics flair—seed set to **137** and I folded it into the training cell so you never forget it mid-sprint. Then I've got a Bash structure.sh that builds the **exact** tree you specified (including the .cmd placeholders and an untagged py_touch.py in each scripts/ folder).

Updated training cell (seed inside)

```
"""
Compile the model and run a short training with early stopping.
"""


from tensorflow.keras import callbacks


def compile_model(model, lr=1e-3, multi_label=False):
  """
  Compile a Keras model for single-label (logits) or multi-label classification.
  """
  if multi_label:
    loss = tf.keras.losses.BinaryCrossentropy()
    metrics = [tf.keras.metrics.BinaryAccuracy(name="bin_acc")]
  else:
    loss = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True)
    metrics = [tf.keras.metrics.SparseCategoricalAccuracy(name="acc")]
  model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=lr),
    loss=loss,
    metrics=metrics
  )
  return model

def train_quick(ds_tr, ds_val, epochs=8):
  """
  Train with EarlyStopping and return (history, seconds_elapsed).
  """
  # Ensure reproducibility seed is set right before training
  set_repro(137)
  es = callbacks.EarlyStopping(monitor="val_loss", patience=3, restore_best_weights=True)
  start = time.time()
  hist = model.fit(ds_tr, validation_data=ds_val, epochs=epochs, callbacks=[es], verbose=1)
  secs = time.time() - start
  return hist, secs

# build + compile
model = build_model()
model = compile_model(model, lr=1e-3, multi_label=False)

history, secs = train_quick(ds_tr, ds_val, epochs=8)
if humanfriendly:
  print("Train time:", humanfriendly.format_timespan(secs))
else:
  print(f"Train time: {secs:.1f}s")
```

PowerShell (reference: working inline “touch”)

```
# structure.ps1 (reference snippet you already verified)
# Create tagged file safely (idempotent)
$dir = Split-Path -Parent $tagged
if (-not (Test-Path $dir)) {
  New-Item -ItemType Directory -Path $dir -Force | Out-Null
}
if (-not (Test-Path $tagged)) {
  New-Item -ItemType File -Path $tagged -Force | Out-Null
}
```

Bash structure.sh (matches your desired tree exactly)

```
#!/usr/bin/env bash
# Usage: ./structure.sh <ROOT_DIR> <tag1> [tag2 ...]
# Creates the structure shown in your example, including:
# - README_<tag>.md
# - notebooks/*_<tag>.ipynb
# - scripts/{py_build_model,py_train_model,py_inference,py_utils}_<tag>.py
# - scripts/{build_model,train_model,inference}_<tag>.cmd
# - scripts/py_touch.py        # untagged (per tag directory)
# - datasets/, models/, logs/, visualizations/, outputs/{csv_logs,gradcam_images}

set -euo pipefail

ROOT_DIR="${1:?Usage: $0 <ROOT_DIR> <tag1> [tag2 ...]}"
shift
if [ $# -lt 1 ]; then
  echo "Need at least one tag (e.g., p_01 p_02)" >&2
  exit 2
```

```
fi

mkd() { mkdir -p -- "$1"; }
touch_safe() {
  local path="$1"
  mkd "$(dirname -- "$path")"
  [ -f "$path" ] || : > "$path"
}

# Per-tag file stems (relative to the tag root)
README_STEM="README.md"

# Notebooks (placed under notebooks/, with tag suffix)
NB_FILES=(
  "notebooks/00_data_exploration.ipynb"
  "notebooks/01_model_build.ipynb"
  "notebooks/02_training.ipynb"
  "notebooks/03_inference_quick_explore.ipynb"
)

# Scripts (tag-suffixed python files)
PY_FILES=(
  "scripts/py_build_model.py"
  "scripts/py_train_model.py"
  "scripts/py_inference.py"
  "scripts/py_utils.py"
)

# Scripts (tag-suffixed CMD placeholders, as requested)
CMD_FILES=(
  "scripts/build_model.cmd"
  "scripts/train_model.cmd"
  "scripts/inference.cmd"
)

# Common per-tag untagged helper
UNTAGGED_COMMON="scripts/py_touch.py"

# Fixed directory set under each tag
DIRS=(
  "notebooks"
  "datasets"
  "models"
  "logs"
  "visualizations"

  "scripts"

  "outputs/csv_logs"

  "outputs/gradcam_images"

)


for tag in "$@"; do

  TAG_DIR="${ROOT_DIR%/}/$tag"


  # Directories

  for d in "${DIRS[@]}"; do

    mkd "$TAG_DIR/$d"

  done


  # README_<tag>.md at tag root

  touch_safe "$TAG_DIR/${README_STEM%.md}_${tag}.md"


  # Notebooks with _<tag>.ipynb inside notebooks/

  for f in "${NB_FILES[@]}"; do

    base="$(basename -- "$f")"

    stem="${base%.*}"

    ext="${base##*.}"
    touch_safe "$TAG_DIR/notebooks/${stem}_${tag}.${ext}"
  done

  # Python scripts with _<tag>.py
  for f in "${PY_FILES[@]}"; do
    base="$(basename -- "$f")"
    stem="${base%.*}"
    ext="${base##*.}"
    touch_safe "$TAG_DIR/scripts/${stem}_${tag}.${ext}"
  done

  # CMD placeholders with _<tag>.cmd
  for f in "${CMD_FILES[@]}"; do
    base="$(basename -- "$f")"
    stem="${base%.*}"
    ext="${base##*.}"
    touch_safe "$TAG_DIR/scripts/${stem}_${tag}.${ext}"
  done

  # Untagged common helper
  touch_safe "$TAG_DIR/$UNTAGGED_COMMON"
done
```

```
echo "Scaffold created under: $ROOT_DIR"
```

Quick test (Linux/AWS)

```
chmod +x structure.sh

TMPDIR="$PWD/test_project"

./structure.sh "$TMPDIR" p_01 p_02

tree -a -F "$TMPDIR"
```

That tree should match your target layout (including .cmd files and py_touch.py untagged within each scripts/). If anything is off by a hair, ping me the tree diff and I'll tweak the script in-place.

You said:

I like the justification you suggested for the use of 137 as a seed. x D I realized that there were some problems with the PowerShell structure.ps1 we built before. <strike>I'll manually do a comm-type comparison with the output from before on the left, common things in the middle, and output from what's desired on the right.</strike> The three lines wouldn't fit, and anyway, they destroyed the easily-readable tree structure. Here's what we had before.

plain

```
|
\---test_project
    \---p_01
        |   00_data_exploration_p_01.ipynb
        |   01_model_build_p_01.ipynb
        |   02_training_p_01.ipynb
        |   03_inference_quick_explore_p_01.ipynb
        |   build_model_p_01.ps1
        |   inference_p_01.ps1
        |   py_build_model_p_01.py
        |   py_inference_p_01.py
        |   py_train_model_p_01.py
        |   py_utils_p_01.py
        |   README_p_01.md
        |   train_model_p_01.ps1
        |
        +---datasets
        +---logs
        +---models
        +---notebooks
        +---outputs
        |   +---csv_logs
        |   \---gradcam_images
        +---scripts
        \---visualizations
```

Here's how I would have liked it to be (with the same files, but some other sub-directories). And, here, I remember why I like the --charset=ascii option (or /a option for Windows). I can write a new tree without going crazy with copy/paste; I just use the keyboard.

plain

```
|
\---test_project
```

```
\---p_01
    |   00_data_exploration_p_01.ipynb
    |   01_model_build_p_01.ipynb
    |   02_training_p_01.ipynb
    |   03_inference_quick_explore_p_01.ipynb
    |   build_model_p_01.ps1
    |   inference_p_01.ps1
    |   py_build_model_p_01.py
    |   py_inference_p_01.py
    |   py_train_model_p_01.py
    |   py_utils_p_01.py
    |   README_p_01.md
    |   train_model_p_01.ps1
    |
    +---datasets
    +---logs
    +---models
    +---notebooks
    +---outputs
    |   +---csv_logs
    |   \---gradcam_images
    +---scripts
    \---visualizations
```

ChatGPT said:

I see what happened — the two trees you pasted are actually identical in content and structure, but your *expectation* vs. *actual* difference was really about **where certain files and subdirectories ended up being created in structure.ps1**.

Specifically:

- In the **“before” tree**, the tagged files (e.g., py_build_model_p_01.py) and tagged .ps1 scripts were created **directly under p_01/**, *alongside* the README and notebooks.
- In your **“desired” tree**, those same files should instead live **inside scripts/** (along with the helper, py_touch.py), with notebooks/ containing only the .ipynb files.

That subtle placement difference is what makes the tree “feel” wrong even though the file list looks the same at first glance.

How to Fix structure.ps1

You just need to adjust the loop that appends _p_XX to filenames so that:

- Notebook files (*.ipynb) are created inside notebooks/
- Script files (*.py, *.ps1) are created inside scripts/
- Only the README lives at the root of each tag dir

Key edit

Change the tagged file generation logic:

```
# Files to create
$Files = @(
  "README.md",
  "notebooks\00_data_exploration.ipynb",
  "notebooks\01_model_build.ipynb",
  "notebooks\02_training.ipynb",
  "notebooks\03_inference_quick_explore.ipynb",
  "scripts\py_build_model.py",
  "scripts\build_model.ps1",
  "scripts\py_train_model.py",
  "scripts\train_model.ps1",
  "scripts\py_inference.py",
  "scripts\inference.ps1",
  "scripts\py_utils.py"
)

foreach ($tag in $Tags) {
  $TagDir = Join-Path $RootDir $tag

  # Create directories first
  $Dirs = @("notebooks", "datasets", "models", "logs", "scripts", "visualizations", "outputs\csv_logs", "outputs\gradcam_images")
  foreach ($d in $Dirs) { New-Item -ItemType Directory -Path (Join-Path $TagDir $d) -Force | Out-Null }

  foreach ($f in $Files) {
    $base   = [System.IO.Path]::GetFileNameWithoutExtension($f)
    $ext    = [System.IO.Path]::GetExtension($f)
    $subdir = Split-Path $f -Parent
    $tagged = if ($subdir) { Join-Path $TagDir "$subdir\$base`_$tag$ext" } else { Join-Path $TagDir "$base`_$tag$ext" }


    $dir = Split-Path -Parent $tagged
    if (-not (Test-Path $dir)) { New-Item -ItemType Directory -Path $dir -Force | Out-Null }
    if (-not (Test-Path $tagged)) { New-Item -ItemType File -Path $tagged -Force | Out-Null }
  }
}
```

✅ Resulting Tree (PowerShell version)

After the change, tree /a /f will give you:

```
\---test_project
    \---p_01
        |   README_p_01.md
        |
        +---datasets
        +---logs
        +---models
        +---notebooks
        |       00_data_exploration_p_01.ipynb
        |       01_model_build_p_01.ipynb
        |       02_training_p_01.ipynb
        |       03_inference_quick_explore_p_01.ipynb
        +---outputs
        |   +---csv_logs
        |   \---gradcam_images
        +---scripts
        |       py_build_model_p_01.py
        |       build_model_p_01.ps1
        |       py_train_model_p_01.py
        |       train_model_p_01.ps1
        |       py_inference_p_01.py
        |       inference_p_01.ps1
        |       py_utils_p_01.py
        \---visualizations
```

This matches the **desired layout** you sketched (scripts grouped under scripts/, notebooks under notebooks/, README at the root).

👉 Do you want me to hand you the **full corrected structure.ps1** (with these fixes already integrated), so you can just drop it in and test without editing?

You said:

Oops, I didn't give the good version. But you figured it out. I figured out how to fix it, and I'll show you the bettered output. However, I was expecting there to be another directory with p_02, i.e. there would be another part of the tree (another subdirectory of test_project) beginning:

powershell

```
  \---p_02
```

```
    |---notebooks
    |   00_data_exploration_p_02.ipynb
    |   01_model_build_p_02.ipynb
```

The bash code (run with chmod a+x structure.sh, then TMPDIR="$PWD/test_project_bash", and finally ./structure.sh "$TMPDIR" p_01 p_02 gave the output as expected. Let me send my improvements to the PS1 script as well as the results I obtain. With this, the part of the tree I send will have the directories: test_project (showing the PowerShell result from before lunch), test_project_bash (containing the bash result from just now), and test_project_ps (containing the PowerShell results with my improvements. The directory structure will be sent in one code block, the contents of my new structure.ps1 in another. They will both come in the next prompt.

ChatGPT said:

Got it—send the two blocks when you're ready. In the meantime, here are quick, targeted tips so your next PS run definitely produces both p_01 **and** p_02:

Why only p_01 might have appeared

- **Argument binding quirk:** If the call ended up passing a **single string** (e.g., "p_01 p_02"), PowerShell treated it as one item.
- **Comma vs. space:** -Tags p_01, p_02 and -Tags p_01 p_02 both *should* work, but mixing quotes/shells/-File can sometimes yield a single token.

Make -Tags robust (accept space- or comma-separated)

Drop this near the top of structure.ps1, right after param(...):

```
# Normalize tags: accept -Tags p_01 p_02  OR  -Tags 'p_01,p_02'
if ($null -ne $Tags) {
  $Tags = (($Tags -join ' ') -split '[,\s]+' | Where-Object { $_ -ne '' }) | Select-Object -Unique
} else {
  $Tags = @('default')
}
```

That coerces whatever you pass into a clean string array.

Known-good invocation forms

```
# 1) Separate tokens (most natural)

.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01 p_02


# 2) Comma-separated

.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01,p_02


# 3) Explicit array

.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags @('p_01','p_02')


# 4) Via -File with bypass (your practice mode)

powershell -ExecutionPolicy Bypass -File .\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01 p_02
```

Tiny debug you can remove later

```
Write-Host "Tags parsed:" ($Tags -join ', ')

Write-Host "Count:" $Tags.Count
```

Optional parity with Bash layout

If you want the **untagged** helper like in Bash, add:

```
$untaggedHelper = Join-Path $TagDir 'scripts\py_touch.py'

if (-not (Test-Path $untaggedHelper)) {

  New-Item -ItemType File -Path $untaggedHelper -Force | Out-Null

}
```

Quick success check

After running:

```
tree /a /f "$PWD\test_project_ps"
```

You should see:

```
\---test_project_ps
    +---p_01
    \---p_02
```

with each tag containing notebooks/, scripts/ (tagged .py and .ps1 or .cmd as you want), datasets/, models/, logs/, visualizations/, and outputs/{csv_logs,gradcam_images}.

Drop your new structure.ps1 and the tree output when ready—I’ll spot-check and, if needed, give you a one-line patch to finish it.

You said:

I like that solution. Can you merge the more robust -Tags code it with my half-way version? I've added some things, including a py_touch.py that isn't used in the structure script, but gives Windows users something similar to touch, as well as giving, e.g. *NIX users with only the base installed and thus without touch(1) an option to create other files easily. Here is my half-way PowerShell version

powershell

```
param(
  [string]$RootDir = "$PWD\new_experiment_dir",
  [string[]]$Tags = @("default")
)


# Ensure root directory exists
if (-not (Test-Path $RootDir)) { New-Item -ItemType Directory -Path $RootDir | Out-Null }
```

```
#out###  No helper required; this was just paralleling our CMD pattern,
#out###+ istelf set up to avoid the obstruse `$ >> filename 2>nul`
#out## Create a helper script for touching files
#out#$TouchScript = Join-Path $RootDir "scripts\ps_touch.ps1"
#out#if (-not (Test-Path $TouchScript)) {
#out#  New-Item -ItemType Directory -Path (Split-Path $TouchScript) -Force | Out-Null
#out#  @'
#out#  param([string[]]$Paths)
#out#  foreach ($p in $Paths) {
#out#    $dir = Split-Path -Parent $p
#out#    if (-not (Test-Path $dir)) {
#out#      New-Item -ItemType Directory -Path $dir -Force | Out-Null
#out#    }
#out#    if (-not (Test-Path $p)) {
#out#      New-Item -ItemType File -Path $p -Force | Out-Null
#out#    }
#out#  }
#out#  '@ | Set-Content -LiteralPath $TouchScript -Encoding UTF8
#out#}

# Files to create
$Files = @(
  "README.md",

  "notebooks\00_data_exploration.ipynb",

  "notebooks\01_model_build.ipynb",

  "notebooks\02_training.ipynb",

  "notebooks\03_inference_quick_explore.ipynb",

  "scripts\py_build_model.py",

  "scripts\build_model.ps1",

  "scripts\py_train_model.py",

  "scripts\train_model.ps1",

  "scripts\py_inference.py",

  "scripts\inference.ps1",

  "scripts\py_utils.py"

)


$UntaggedCommon="scripts/py_touch.py"


foreach ($tag in $Tags) {

  $TagDir = Join-Path $RootDir $tag


  # Create main directories

  $Dirs = @("notebooks", "datasets", "models", "logs", "scripts", "visualizations", "outputs\csv_logs", "outputs\gradcam_images")

  foreach ($d in $Dirs) { New-Item -ItemType Directory -Path (Join-Path $TagDir $d) -Force | Out-Null }
```

```
# Create files with tag appended to stem
foreach ($f in $Files) {
  $relpath = Split-Path -Path $f -Parent
  $base    = [System.IO.Path]::GetFileNameWithoutExtension($f)
  $ext     = [System.IO.Path]::GetExtension($f)
  $tagged  = Join-Path -Path (Join-Path -Path $TagDir -ChildPath $relpath) -ChildPath "$base`_$tag$ext"
  #$tagged  = Join-Path $TagDir "$relpath" "$base`_$tag$ext"

  $dir = Split-Path -Parent $tagged
  if (-not (Test-Path $dir)) {
    New-Item -ItemType Directory -Path $dir -Force | Out-Null
  }
  if (-not (Test-Path $tagged)) {
    New-Item -ItemType File -Path $tagged -Force | Out-Null
  }
}

##  I _do_ include a per-experiment helper, written in Python, in
##+ case some kind of `touch' functionality be desired that's
##+ consistent between Windows and Linux (*NIX). This is the sole
##+ member of the category, UNTAGGED_COMMON, in `structure.sh'.
```

```
  # Create (untagged common) py_touch.py if missing
  $PyTouchPath = Join-Path $TagDir "scripts\py_touch.py"
  if (-not (Test-Path $PyTouchPath)) {
    @'
import sys
from pathlib import Path
for f in sys.argv[1:]: Path(f).touch(exist_ok=True)

'@ | Set-Content -Path $PyTouchPath
  }

  #Info on the helper
  Write-Host "  -------------------------------------------------------"
  Write-Host "  OS-agnostic helper,"
  Write-Host "  $PyTouchPath"
  Write-Host "  provided for tag, ``$tag', in case it be desired."
  Write-Host "  This could prove immensely helpful for Windows users."
  Write-Host "  On *NIX-type systems, I suggest using ``touch(1)', unless"
  Write-Host "  it be not installed, e.g. unless you have only the base"
  Write-Host "  installation."
  Write-Host ""
```

```
}


Write-Host "----------------------------------------------------------------"
Write-Host "Project scaffolding with tags and subdirectories created at $RootDir"
Write-Host "----------------------------------------------------------------"
```

After running, note that

powershell

```
PS C:\David\my_repos_dwb\fhtw-paper-code-prep> cat .\test_project_ps\p_01\scripts\py_touch.py
import sys
from pathlib import Path
for f in sys.argv[1:]: Path(f).touch(exist_ok=True)


PS C:\David\my_repos_dwb\fhtw-paper-code-prep>
```

I can send the improved bash version and the tree outputs, if desired.

ChatGPT said:

Thought for 7s

Absolutely—here's your **merged structure.ps1** with the robust -Tags normalization dropped into your "half-way" version, plus a couple of tiny niceties (UTF-8 when writing py_touch.py, and a short, optional debug echo for the tags).

Changes from your snippet:

- Normalize -Tags so it accepts p_01 p_02, p_01,p_02, or @('p_01','p_02').
- Keep your untagged scripts\py_touch.py helper exactly as you wrote it.
- Leave everything else as-is (files stay under notebooks/ and scripts/, README at tag root).

```
param(
  [string]$RootDir = "$PWD\new_experiment_dir",
  [string[]]$Tags  = @("default")
)

# Normalize tags: accept space- or comma-separated, or explicit arrays.
if ($null -ne $Tags) {
  if ($Tags -is [string]) { $Tags = @($Tags) }
  $Tags = (($Tags -join ' ') -split '[,\s]+' | Where-Object { $_ -ne '' }) | Select-Object -Unique
} else {
  $Tags = @('default')
}

# (Optional) quick debug; comment out when not needed
# Write-Host "Tags parsed:" ($Tags -join ', ')

# Ensure root directory exists
if (-not (Test-Path $RootDir)) {
  New-Item -ItemType Directory -Path $RootDir | Out-Null
}

# Files to create (relative to each tag dir)
```

```
$Files = @(
  "README.md",
  "notebooks\00_data_exploration.ipynb",
  "notebooks\01_model_build.ipynb",
  "notebooks\02_training.ipynb",
  "notebooks\03_inference_quick_explore.ipynb",
  "scripts\py_build_model.py",
  "scripts\build_model.ps1",
  "scripts\py_train_model.py",
  "scripts\train_model.ps1",
  "scripts\py_inference.py",
  "scripts\inference.ps1",
  "scripts\py_utils.py"
)

$UntaggedCommon = "scripts\py_touch.py"

foreach ($tag in $Tags) {
  $TagDir = Join-Path $RootDir $tag

  # Create main directories
  $Dirs = @(
    "notebooks", "datasets", "models", "logs", "scripts", "visualizations",
    "outputs\csv_logs", "outputs\gradcam_images"
  )
  foreach ($d in $Dirs) {
    New-Item -ItemType Directory -Path (Join-Path $TagDir $d) -Force | Out-Null
  }

  # Create files with tag appended to stem, preserving subdir
  foreach ($f in $Files) {
    $relpath = Split-Path -Path $f -Parent
    $base    = [System.IO.Path]::GetFileNameWithoutExtension($f)
    $ext     = [System.IO.Path]::GetExtension($f)
    $destDir = if ($relpath) { Join-Path $TagDir $relpath } else { $TagDir }
    $tagged  = Join-Path $destDir "$base`_$tag$ext"

    $dir = Split-Path -Parent $tagged
    if (-not (Test-Path $dir)) {
      New-Item -ItemType Directory -Path $dir -Force | Out-Null
    }
    if (-not (Test-Path $tagged)) {
      New-Item -ItemType File -Path $tagged -Force | Out-Null
    }
  }

  # Per-experiment, untagged helper for touch-like behavior (Windows/*NIX)
  $PyTouchPath = Join-Path $TagDir $UntaggedCommon
  if (-not (Test-Path $PyTouchPath)) {
    @'
import sys
from pathlib import Path
for f in sys.argv[1:]:
  Path(f).touch(exist_ok=True)
'@ | Set-Content -Path $PyTouchPath -Encoding UTF8
  }

  # Info on the helper
  Write-Host "  ----------------------------------------------------------"
  Write-Host "  OS-agnostic helper:"
  Write-Host "  $PyTouchPath"
  Write-Host "  provided for tag '$tag'."
  Write-Host "  On *NIX, prefer touch(1) when available; this is a fallback."
  Write-Host ""
}
```

```
Write-Host "------------------------------------------------------------------"

Write-Host "Project scaffolding with tags and subdirectories created at $RootDir"

Write-Host "------------------------------------------------------------------"
```

Known-good invocations

```
# Separate tokens

.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01 p_02



# Comma-separated

.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01,p_02



# Explicit array

.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags @('p_01','p_02')



# Via -File (your practice mode)

powershell -ExecutionPolicy Bypass -File .\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01 p_02
```

If you want, send your improved Bash version and the tree outputs—happy to sanity-check parity and tweak anything so AWS matches Windows down to the path separators and filenames.

You said:

I love it! Oh, I need to mention the differences with what was there as CMD scripts. You'll likely see it with my improved bash version. (I want any invocation of structure.{sh/ps1} to give the same file tree. The idea is that, e.g. py_build_model.py, build_model.ps1, and build_model.sh should give us equivalent results. It seems that you suggested build_model.{sh/ps1}. As we go to AWS, we'll just write one build_model.<whichever is best practice> for our *Done not perfect* run. Maybe I'll get back to the others, maybe not. Program structure.sh

bash

```
#!/usr/bin/env bash

# Usage: ./structure.sh <ROOT_DIR> <tag1> [tag2 ...]

# Creates the structure shown in your example, including:

# - README_<tag>.md

# - notebooks/*_<tag>.ipynb

# - scripts/{py_build_model,py_train_model,py_inference,py_utils}_<tag>.py

# - scripts/{build_model,train_model,inference}_<tag>.cmd

# - scripts/py_touch.py        # untagged (per tag directory)

# - datasets/, models/, logs/, visualizations/, outputs/{csv_logs,gradcam_images}


set -euo pipefail


ROOT_DIR="${1:?Usage: $0 <ROOT_DIR> <tag1> [tag2 ...]}"

shift

if [ $# -lt 1 ]; then

  echo "Need at least one tag (e.g., p_01 p_02)" >&2

  exit 2

fi


mkd() { mkdir -p -- "$1"; }

touch_safe() {

  local path="$1"
  mkd "$(dirname -- "$path")"
  [ -f "$path" ] || : > "$path"
}

# Per-tag file stems (relative to the tag root)
README_STEM="README.md"

# Notebooks (placed under notebooks/, with tag suffix)
NB_FILES=(
  "notebooks/00_data_exploration.ipynb"
  "notebooks/01_model_build.ipynb"
  "notebooks/02_training.ipynb"
  "notebooks/03_inference_quick_explore.ipynb"
)

#  Scripts (tag-suffixed python placeholders)
#+ These could perform the same thing as the
#+ *.sh versions and the *.ps1 versions.
#+ We'll probably only get to one of them
#+ before the Fragmentology publication.
PY_FILES=(
  "scripts/py_build_model.py"
  "scripts/py_train_model.py"
  "scripts/py_inference.py"
  "scripts/py_utils.py"
)

#  Scripts (tag-suffixed SH placeholders)
#+ These could perform the same thing as the
#+ py_*.py versions and the *.ps1 versions.
#+ We'll probably only get to one of them
#+ before the Fragmentology publication.
THE_SH_FILES=(
  "scripts/build_model.sh"
  "scripts/train_model.sh"
  "scripts/inference.sh"
)

#  Scripts (tag-suffixed SH placeholders)
#+ These could perform the same thing as the
#+ py_*.py versions and the *.ps1 versions.
#+ We'll probably only get to one of them
#+ before the Fragmentology publication.
THE_PS_FILES=(
  "scripts/build_model.ps1"
  "scripts/train_model.ps1"
  "scripts/inference.ps1"
)

# Common per-tag untagged helper
UNTAGGED_COMMON="scripts/py_touch.py"

# Fixed directory set under each tag
DIRS=(
  "notebooks"
  "datasets"
  "models"
  "logs"
  "visualizations"
  "scripts"
  "outputs/csv_logs"
  "outputs/gradcam_images"
)

for tag in "$@"; do
  TAG_DIR="${ROOT_DIR%/}/$tag"
```

```
# Directories

for d in "${DIRS[@]}"; do

  mkd "$TAG_DIR/$d"

done


# README_<tag>.md at tag root

touch_safe "$TAG_DIR/${README_STEM%.md}_${tag}.md"


# Notebooks with _<tag>.ipynb inside notebooks/

for f in "${NB_FILES[@]}"; do

  base="$(basename -- "$f")"

  stem="${base%.*}"

  ext="${base##*.}"

  touch_safe "$TAG_DIR/notebooks/${stem}_${tag}.${ext}"

done


# Python scripts with _<tag>.py

for f in "${PY_FILES[@]}"; do

  base="$(basename -- "$f")"

  stem="${base%.*}"

  ext="${base##*.}"
```

```
    touch_safe "$TAG_DIR/scripts/${stem}_${tag}.${ext}"
  done

  # CMD placeholders with _<tag>.cmd
  for f in "${CMD_FILES[@]}"; do
    base="$(basename -- "$f")"
    stem="${base%.*}"
    ext="${base##*.}"
    touch_safe "$TAG_DIR/scripts/${stem}_${tag}.${ext}"
  done

  # Untagged common helper
  touch_safe "$TAG_DIR/$UNTAGGED_COMMON"

  ##  The idea of a `touch'-like function for PowerShell was
  ##+ abandoned. I _do_ include a per-experiment helper, written
  ##+ in Python, in case some kind of `touch' functionality be
  ##+ desired that's consistent between Windows and Linux (*NIX)
  py_touch_path="$TAG_DIR/$UNTAGGED_COMMON"
  if [ ! -f "$py_touch_path" ]; then
    cat << 'EOF' > "$py_touch_path"
import sys
```

```
from pathlib import Path
for f in sys.argv[1:]: Path(f).touch(exist_ok=True)
EOF
  fi

  #Info on the helper
  echo "  ------------------------------------------------------------------"
  echo "  OS-agnostic helper,"
  echo "  $PyTouchPath"
  echo "  provided for tag, ``$tag', in case it be desired."
  echo "  This could prove immensely helpful for Windows users."
  echo "  On *NIX-type systems, I suggest using ``touch(1)', unless"
  echo "  it be not installed, e.g. unless you have only the base"
  echo "  installation."
  echo

done

echo "-------------------------------------------------------------------------------"
echo "Project scaffolding with tags and subdirectories created at $ROOT_DIR"
echo "-------------------------------------------------------------------------------"
```

Please wait to comment until I put the output of the bash version, which will come in the next prompt.

ChatGPT said:

Thought for a few seconds

Got it — I'll hold comments until you post the Bash tree output. When you're ready, drop it in and I'll check parity and suggest any tiny patch needed.

You said:

```
Anast@DESKTOP-O7KM5A5 /cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep $ rm -rf test_project_bash/ Anast@DESKTOP-O7KM5A5
/cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep $ TMPDIR="$PWD/test_project_bash" Anast@DESKTOP-O7KM5A5
/cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep $ ./structure.sh "$TMPDIR" p_01 p_02 --------------------------------------------------------- OS-
agnostic helper, /cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep/test_project_bash/p_01/scripts/py_touch.py provided for tag, p_01', in
case it be desired. This could prove immensely helpful for Windows users. On *NIX-type systems, I suggest using touch(1)', unless it be not installed,
e.g. unless you have only the base installation. --------------------------------------------------------- OS-agnostic helper,
/cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep/test_project_bash/p_02/scripts/py_touch.py provided for tag, p_02', in case it be desired.
This could prove immensely helpful for Windows users. On *NIX-type systems, I suggest using touch(1)', unless it be not installed, e.g. unless you
have only the base installation. -------------------------------------------------------------------- Project scaffolding with tags and subdirectories created at
/cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep/test_project_bash -------------------------------------------------------------------- Anast@DESKTOP-
O7KM5A5 /cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep $ tree --charset=ascii -F . . |-- LICENSE* |-- NOTWORKING_structure.bat* |--
Paper_Code_Prep_01-00-00_CNN_GradCAM_TF.ipynb* |-- README.md* |-- cmd_for_notebook_local_and_pre_aws_env.log* |--
dataset_preparation_examples/ | |-- #collapsed#|-- environment_vanillacnn.yml* |-- experiment_environment_examples/ |--
general_lab_notebooks_-_other_examples/ | |-- lab_notebook_1756385640_2025-08-28T125400-0600.md* | |--
lab_notebook_bballdave025_1756554507_2025-08-30T124827-0500.docx* | |-- lab_notebook_bballdave025_1756554507_2025-08-30T124827-
0500.md* | |-- ~$b_notebook_bballdave025_1756554507_2025-08-30T124827-0500.docx* | -- ~WRL3165.tmp* |-- img/ | |--
BW_post_from_905_col_terminal.png* | |-- BW_pre_conv_from_903_col_terminal_-_Best.png* | |-- README.md* | |--
WinFileExplorer_bitdepth_gs_post_conv.png* | |-- WinFileExplorer_bitdepth_gs_pre_conv.png* | |-- consistentized_3331_dataset.txt* | |--
fs_watermarks_etc_pre_conversion_1753209154_2025-07-22T123234-0600.out* | |-- seeofit1_edit.png* | |-- seeofit1_edit_crop.png* | |--
seeofit2_edit.png* | -- seeofit2_edit_crop.png* |-- structure.ps1* |-- structure.sh* |-- test_project_bash/ | |-- p_01/ | | |-- README_p_01.md* | |
|-- datasets/ | | |-- logs/ | | |-- models/ | | |-- notebooks/ | | | |-- 00_data_exploration_p_01.ipynb* | | | |-- 01_model_build_p_01.ipynb* | | |
|-- 02_training_p_01.ipynb* | | | -- 03_inference_quick_explore_p_01.ipynb* | | |-- outputs/ | | | |-- csv_logs/ | | | -- gradcam_images/ | | |--
scripts/ | | | |-- py_build_model_p_01.py* | | | |-- py_inference_p_01.py* | | | |-- py_touch.py* | | | |-- py_train_model_p_01.py* | | | --
py_utils_p_01.py* | | -- visualizations/ | -- p_02/ | |-- README_p_02.md* | |-- datasets/ | |-- logs/ | |-- models/ | |-- notebooks/ | | |--
00_data_exploration_p_02.ipynb* | | |-- 01_model_build_p_02.ipynb* | | |-- 02_training_p_02.ipynb* | | --
```

03_inference_quick_explore_p_02.ipynb* | |-- outputs/ | | |-- csv_logs/ | | -- gradcam_images/ | |-- scripts/ | | |-- py_build_model_p_02.py* | | |-- py_inference_p_02.py* | | |-- py_touch.py* | | |-- py_train_model_p_02.py* | | -- py_utils_p_02.py* | -- visualizations/ |-- test_project_ps/ | -- p_01/ | |-- README_p_01.md* | |-- datasets/ | |-- logs/ | |-- models/ | |-- notebooks/ | | |-- 00_data_exploration_p_01.ipynb* | | |-- 01_model_build_p_01.ipynb* | | |-- 02_training_p_01.ipynb* | | -- 03_inference_quick_explore_p_01.ipynb* | |-- outputs/ | | |-- csv_logs/ | | -- gradcam_images/ | |-- scripts/ | | |-- build_model_p_01.ps1* | | |-- inference_p_01.ps1* | | |-- py_build_model_p_01.py* | | |-- py_inference_p_01.py* | | |-- py_touch.py* | | |-- py_train_model_p_01.py* | | |-- py_utils_p_01.py* | | -- train_model_p_01.ps1* | -- visualizations/ -- validate_env.py* 36 directories, 82 files Anast@DESKTOP-O7KM5A5 /cygdrive/c/David/my_repos_dwb/fhtw-paper-code-prep $

ChatGPT said:

Thought for 34s

Nice run! I can see exactly what's happening from that tree:

- **Bash**: ✅ both p_01 and p_02 created, but only the **py_*.py** files showed up under scripts/. The .sh and .ps1 placeholders didn't appear.
- **PowerShell**: ✅ structure correct for **p_01**, but **p_02** is missing (either tag parsing or invocation) — and you've got the **.ps1** files there, not .sh.

Below are **tiny, surgical patches** so both scripts produce the **same tree**: py_*.py + *.sh + *.ps1 placeholders under scripts/ for every tag.

Patch 1 — Bash: add loops for .sh + .ps1, fix helper echo

In your structure.sh, you already defined:

- THE_SH_FILES=(scripts/build_model.sh scripts/train_model.sh scripts/inference.sh)
- THE_PS_FILES=(scripts/build_model.ps1 scripts/train_model.ps1 scripts/inference.ps1)

But you never looped them (and the echo uses a wrong var name). Add these loops **after** the Python loop, and fix the echo:

```
 # Shell scripts with _<tag>.sh
 for f in "${THE_SH_FILES[@]}"; do
   base="$(basename -- "$f")"
   stem="${base%.*}"
```

```
  ext="${base##*.}"
  touch_safe "$TAG_DIR/scripts/${stem}_${tag}.${ext}"
done

# PowerShell scripts with _<tag>.ps1
for f in "${THE_PS_FILES[@]}"; do
  base="$(basename -- "$f")"
  stem="${base%.*}"
  ext="${base##*.}"
  touch_safe "$TAG_DIR/scripts/${stem}_${tag}.${ext}"
done

# Untagged common helper
touch_safe "$TAG_DIR/$UNTAGGED_COMMON"

# Info on the helper (fix variable name)
echo "  --------------------------------------------------------"
echo "  OS-agnostic helper,"
echo "  $py_touch_path"
echo "  provided for tag '$tag'."
echo "  This could prove immensely helpful for Windows users."
echo "  On *NIX-type systems, prefer touch(1) when available."
```

```
  echo
```

Why this matters: your earlier version had a stray reference to CMD_FILES and printed $PyTouchPath (capital P), which isn’t defined — so the message showed the literal string.

Quick re-test

```
rm -rf test_project_bash

TMPDIR="$PWD/test_project_bash"

./structure.sh "$TMPDIR" p_01 p_02

tree --charset=ascii -F "$TMPDIR"
```

You should now see, for each tag:

```
scripts/
  py_build_model_<tag>.py
  py_train_model_<tag>.py
  py_inference_<tag>.py
  py_utils_<tag>.py
  build_model_<tag>.sh
  train_model_<tag>.sh
  inference_<tag>.sh
  build_model_<tag>.ps1
  train_model_<tag>.ps1
  inference_<tag>.ps1
  py_touch.py
```

Patch 2 — PowerShell: ensure both tags + add .sh placeholders

You asked for parity, so let’s have PS create the .sh placeholders too. Two micro-edits:

1. **(You already liked this)** Normalize -Tags so both p_01 p_02 and p_01,p_02 work:

```
if ($null -ne $Tags) {
  if ($Tags -is [string]) { $Tags = @($Tags) }
  $Tags = (($Tags -join ' ') -split '[,\s]+' | Where-Object { $_ -ne '' }) | Select-Object -Unique
} else {
  $Tags = @('default')
}
```

2. **Add the .sh files** to $Files:

```
$Files = @(
  "README.md",
  "notebooks\00_data_exploration.ipynb",
  "notebooks\01_model_build.ipynb",
  "notebooks\02_training.ipynb",
  "notebooks\03_inference_quick_explore.ipynb",
  "scripts\py_build_model.py",
  "scripts\py_train_model.py",
  "scripts\py_inference.py",
  "scripts\py_utils.py",
  "scripts\build_model.ps1",
  "scripts\train_model.ps1",
  "scripts\inference.ps1",
  "scripts\build_model.sh",
  "scripts\train_model.sh",
  "scripts\inference.sh"
)
```

The rest of your code (where you append _$tag and create in the correct subdir) already handles these; you don’t need to change anything else.

Known-good invocations

```
# (recommended) separate tokens
.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01 p_02


# comma-separated
.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01,p_02


# explicit array
.\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags @('p_01','p_02')


# your practice mode (non-persistent)
powershell -ExecutionPolicy Bypass -File .\structure.ps1 -RootDir "$PWD\test_project_ps" -Tags p_01 p_02
```

Quick check

```
tree /a /f "$PWD\test_project_ps"
```

You should now see **both** p_01 and p_02, each with:

```
scripts\
```

py_build_model_<tag>.py

py_train_model_<tag>.py

py_inference_<tag>.py

py_utils_<tag>.py

build_model_<tag>.ps1

train_model_<tag>.ps1

inference_<tag>.ps1

build_model_<tag>.sh

train_model_<tag>.sh

inference_<tag>.sh

py_touch.py

Tiny cross-platform notes (just so you don't trip later)

- **Line endings:** If you edit .sh on Windows, consider ensuring **LF** endings (dos2unix) before running on Linux. Placeholders are empty, so it won't matter now, but good to know.
- **Executable bit on AWS:** After scaffolding on Linux, remember chmod +x scripts/*_p_*.sh before use.
- **CMD option (optional):** If you later want .cmd placeholders too, we can add a third trio (build_model, train_model, inference) with .cmd extensions to both scripts.

If you paste your updated structure.ps1 and re-run with -Tags p_01 p_02, I'm expecting parity with your Bash run. If anything's off by a filename or path, shoot the new tree and I'll hand you a one-liner fix.

You said:

Shoot, I didn't cut out the extra parts of my directory, which I haven't had before. I think you'll likely want to reanalyze, though what you have now is spot on as far as no loops for SH and PS1 Actually, do you need the more-concise tree output, or does that tell you all you need?

ChatGPT said:

Thought for 14s

No need for a more concise tree—what you posted gave me everything I needed. The key gaps were exactly what you flagged:

- Bash: loops for the .sh and .ps1 placeholders were missing; helper echo used the wrong var name.
- PowerShell: tags normalization (to ensure p_02 appears) and adding .sh placeholders for parity.

You've got the right fixes queued up. To keep things snappy, here's a tiny **parity check** you can run after each script to confirm both produce the *same* structure without scrolling huge trees.

Bash quick-check

```
# Rebuild
rm -rf test_project_bash
TMPDIR="$PWD/test_project_bash"
./structure.sh "$TMPDIR" p_01 p_02

# Verify counts and filenames per tag
for t in p_01 p_02; do
  echo "== $t notebooks ==";  ls -1 "$TMPDIR/$t/notebooks" | sort
  echo "== $t scripts ==";    ls -1 "$TMPDIR/$t/scripts"   | sort
done

# Expected counts:
```

```
# notebooks: 4 files
# scripts:   11 files (4 py_* + 3 *.sh + 3 *.ps1 + py_touch.py)
```

PowerShell quick-check

```
# Rebuild
Remove-Item -Recurse -Force .\test_project_ps -ErrorAction SilentlyContinue
$root = "$PWD\test_project_ps"
.\structure.ps1 -RootDir $root -Tags p_01 p_02

# Verify counts and filenames per tag
foreach ($t in 'p_01','p_02') {
  "`n== $t notebooks =="
  Get-ChildItem -Name "$root\$t\notebooks" | Sort-Object
  "== $t scripts =="
  Get-ChildItem -Name "$root\$t\scripts"   | Sort-Object
}

# Expected counts:
# notebooks: 4 files
# scripts:   11 files (4 py_* + 3 *.sh + 3 *.ps1 + py_touch.py)
```

If those lists match across Bash and PowerShell, you’re in perfect parity for AWS vs. Windows:

- notebooks/ → 4 _p_<tag>.ipynb
- scripts/ → py_*_p_<tag>.py, *_<tag>.sh, *_<tag>.ps1, and py_touch.py

- plus the common directories (datasets/, models/, logs/, visualizations/, outputs/{csv_logs,gradcam_images}) and README_<tag>.md at each tag root.

If anything's off by even a filename, paste just those two small "scripts" listings for p_01 and p_02 from each run, and I'll hand you a one-line patch.

You said:

That sounds great! I need to run to the warehouse, now, but I'll be putting in those suggestions. I was thinking about dos2unix / unix2dos. Do you know of a quick version (or versions) that we could put in a python script, heredoc or echo into structure.{whatever}, and thus remove the need for those line-ending utilities?

ChatGPT said:

Thought for 24s

Totally doable—skip external dos2unix/unix2dos and drop in a tiny **Python EOL normalizer** you can vendor into your scaffolds. Below are:

1. A compact normalize_eol.py (two-space indents + heredoc docstring).
2. How to **emit it** from PowerShell and Bash during scaffolding.
3. One-line **calls** to make .sh → LF and .ps1/.cmd → CRLF automatically.

---

1) normalize_eol.py (drop-in helper)

```
"""
Normalize line endings (EOL) for text files.


Usage modes:
  A) By extension map (recommended)
    python normalize_eol.py --root <DIR> --map sh=lf,ps1=crlf,cmd=crlf,py=lf,ipynb=lf,md=lf


  B) Explicit mode on listed files
```

```
    python normalize_eol.py --to-lf  file1 file2 ...
    python normalize_eol.py --to-crlf file1 file2 ...

Notes:
  - Skips binaries by a simple heuristic (NULL byte check).
  - Only rewrites when a change is needed.
"""

import argparse, os, sys
from pathlib import Path

def is_binary(data: bytes) -> bool:
  """
  Heuristic: treat as binary if there is a NUL byte.
  """
  return b"\x00" in data

def normalize_bytes(data: bytes, mode: str) -> bytes:
  """
  Convert EOLs:
    mode='lf'   -> \n
    mode='crlf' -> \r\n
  """
  # First unify to LF
  text = data.replace(b"\r\n", b"\n").replace(b"\r", b"\n")
  if mode == "lf":
    return text
  elif mode == "crlf":
    return text.replace(b"\n", b"\r\n")
  else:
    raise ValueError(f"Unknown mode: {mode}")

def normalize_file(path: Path, mode: str) -> bool:
  """
  Normalize a single file in-place. Returns True if modified.
  """
  try:
    raw = path.read_bytes()
  except Exception:
    return False
  if is_binary(raw):
    return False
  new = normalize_bytes(raw, mode)
  if new != raw:
    path.write_bytes(new)
    return True
  return False

def parse_map(map_str: str) -> dict:
  """
  Parse 'ext=mode,ext=mode' into dict like {'.sh':'lf', '.ps1':'crlf'}
  """
  out = {}
  for part in map_str.split(","):
    part = part.strip()
    if not part:
      continue
    k, v = part.split("=")
    ext = k.strip().lower()
    if not ext.startswith("."):
      ext = "." + ext
    out[ext] = v.strip().lower()
  return out

def normalize_by_map(root: Path, extmap: dict) -> int:
  """
  Walk root and apply per-extension modes. Returns count of modified files.
  """
  n = 0
  for p in root.rglob("*"):
    if not p.is_file():
      continue
    mode = extmap.get(p.suffix.lower())
    if not mode:
      continue
    if normalize_file(p, mode):
      n += 1
  return n


def main(argv=None):
  """
  CLI entry point.
  """
  ap = argparse.ArgumentParser()
  g = ap.add_mutually_exclusive_group()
  g.add_argument("--to-lf",   action="store_true", help="Force LF on listed files")
  g.add_argument("--to-crlf", action="store_true", help="Force CRLF on listed files")
  ap.add_argument("--root", type=Path, help="Directory to normalize recursively")
  ap.add_argument("--map",  type=str, help="Extension map like 'sh=lf,ps1=crlf'")
  ap.add_argument("files", nargs="*", type=Path, help="Files to normalize (with --to-*)")
  args = ap.parse_args(argv)

  # Mode B: explicit files
  if args.to_lf or args.to_crlf:
    mode = "lf" if args.to_lf else "crlf"
    changed = 0
    for f in args.files:
      if normalize_file(f, mode):
        changed += 1
    print(f"Changed {changed} files.")
    return 0

  # Mode A: by-extension map under --root
  if args.root and args.map:
    extmap = parse_map(args.map)
    changed = normalize_by_map(args.root, extmap)
    print(f"Changed {changed} files under {args.root}.")
    return 0

  ap.error("Provide either (--to-lf|--to-crlf files...) or --root DIR --map ext=mode,...")
```

```
if __name__ == "__main__":
  sys.exit(main())
```

---

2) Emit the helper during scaffolding

PowerShell: write it next to py_touch.py (per tag)

Add after you create py_touch.py:

```
# Create normalize_eol.py (untagged helper)
$NormPath = Join-Path $TagDir "scripts\normalize_eol.py"
if (-not (Test-Path $NormPath)) {
@'
<PASTE normalize_eol.py CONTENT HERE>
'@ | Set-Content -Path $NormPath -Encoding UTF8
}
```

Bash: write it next to py_touch.py (per tag)

Put after you emit py_touch.py:

```
norm_path="$TAG_DIR/scripts/normalize_eol.py"
if [ ! -f "$norm_path" ]; then
  cat > "$norm_path" << 'EOF'
<PASTE normalize_eol.py CONTENT HERE>
EOF
fi
```

(If you prefer a single shared copy, emit it once at the repo root instead of per-tag, and adjust the paths below.)

---

3) One-liners to normalize after scaffolding

Linux/AWS (inside structure.sh, after files are created)

```
# Normalize EOLs: .sh -> LF, .ps1/.cmd -> CRLF, others LF
python3 "$TAG_DIR/scripts/normalize_eol.py" \
  --root "$TAG_DIR" \
  --map 'sh=lf,ps1=crlf,cmd=crlf,py=lf,ipynb=lf,md=lf'
```

Windows/PowerShell (inside structure.ps1, after files are created)

```
# Normalize EOLs: .sh -> LF, .ps1/.cmd -> CRLF, others LF
$NormPath = Join-Path $TagDir "scripts\normalize_eol.py"
# Try python; fall back to py launcher if needed
$py = "python"
try {
  & $py $NormPath --root $TagDir --map "sh=lf,ps1=crlf,cmd=crlf,py=lf,ipynb=lf,md=lf"
} catch {
  & py -3 $NormPath --root $TagDir --map "sh=lf,ps1=crlf,cmd=crlf,py=lf,ipynb=lf,md=lf"
}
```

Notes that save pain later

- **Shebang + CRLF**: #!/usr/bin/env bash must be on a line ending in **LF** or /usr/bin/env will see bash\r. The mapping above ensures all .sh → LF.
- **PowerShell & CRLF**: .ps1 works with LF on modern shells, but CRLF is the Windows-native convention and avoids edge-tool complaints.

- **.cmd**: stick with CRLF on Windows.
- **Binary safety**: the helper skips files containing NULL; your tree mostly has text, but this avoids mangling images, etc.
- **Speed**: it only rewrites files whose EOLs actually change.

If you want, I can also spit out a ready-to-paste **mini block** for your READMEs explaining the EOL policy (“*.sh → LF; *.ps1/*.cmd → CRLF; everything else LF”) so future you (or collaborators) know it’s intentional.

You said:

That's good. I didn't know the word (emit) for the concept of making sure they would be usable. I'll make changes to both bash and PowerShell when I'm back (which will be in about 5 hours) or tomorrow (hopefully before 09:00).

ChatGPT said:

Thought for 10s

Sounds good—have a smooth run to the warehouse. When you’re back, here’s a tight “start-here” so you can pick up momentum fast:

When you get back

1. **Bash parity patch**
    - Add the missing loops for THE_SH_FILES and THE_PS_FILES.
    - Fix the helper echo to use "$py_touch_path".
    - Rebuild + quick check:
    - rm -rf test_project_bash
    - TMPDIR="$PWD/test_project_bash"
    - ./structure.sh "$TMPDIR" p_01 p_02
    - for t in p_01 p_02; do
    -   echo "== $t notebooks =="; ls -1 "$TMPDIR/$t/notebooks" | sort
    -   echo "== $t scripts ==";   ls -1 "$TMPDIR/$t/scripts" | sort

   - done
   - Expect: notebooks=4 files; scripts=11 files (4 py_* + 3 *.sh + 3 *.ps1 + py_touch.py).
2. **PowerShell parity patch**
   - Keep the **Tags normalizer** we merged.
   - Add .sh placeholders to $Files.
   - Rebuild + quick check:
   - Remove-Item -Recurse -Force .\test_project_ps -ErrorAction SilentlyContinue
   - $root = "$PWD\test_project_ps"
   - powershell -ExecutionPolicy Bypass -File .\structure.ps1 -RootDir $root -Tags p_01 p_02
   - foreach ($t in 'p_01','p_02') {
   -   "`n== $t notebooks =="; Get-ChildItem -Name "$root\$t\notebooks" | Sort-Object
   -   "== $t scripts ==";   Get-ChildItem -Name "$root\$t\scripts" | Sort-Object
   - }
3. **Emit EOL normalizer (skip dos2unix/unix2dos)**
   - Drop normalize_eol.py alongside py_touch.py (per tag) in both scripts, then call:
     - Bash:
     - python3 "$TAG_DIR/scripts/normalize_eol.py" \
     -   --root "$TAG_DIR" \
     -   --map 'sh=lf,ps1=crlf,cmd=crlf,py=lf,ipynb=lf,md=lf'
     - PowerShell:
     - & python "$TagDir\scripts\normalize_eol.py" --root $TagDir --map "sh=lf,ps1=crlf,cmd=crlf,py=lf,ipynb=lf,md=lf"
4. **Cygwin/WSL notes**

- After scaffolding on Linux/AWS: chmod +x "$TAG_DIR/scripts/"*"_p_"*.sh
- If Git is in play, consider a tiny .gitattributes so collaborators get the same EOLs:
- *.sh   text eol=lf
- *.ps1 text eol=crlf
- *.cmd text eol=crlf
- *.py   text eol=lf
- *.md   text eol=lf
- *.ipynb text eol=lf

5. **Notebook sprint**
   - Use the updated training cell (with set_repro(137) inside) and aim for quick plots + test eval.
   - If CPU-bound, drop batch_size to 64 or epochs to 4—goal is a clean pass, not SOTA.

If any list differs between Bash vs. PowerShell in those quick checks, paste just the two short scripts/ listings and I'll hand you a one-liner fix.