Assignment Date : 13-08-2025

Name:Lalit Joshi

Email : [Joshilalit2275@gmail.com](mailto:Joshilalit2275@gmail.com)

Userid: **34734**

**Topic 1:** **Process check and Delete in psadt & for loop**

PSAppDeployToolkit (PSADT) allows you to **check for running processes** and **stop (delete) processes** during application deployment to ensure no conflicting processes interfere with installation or uninstallation.

**How to Check Processes:**

* Use PowerShell’s Get-Process cmdlet to find running processes by name or ID.
* Example:

powershell

CopyEdit

Get-Process -Name "Process.exe"

* You can check if a specific process is running by checking if Get-Process returns any result.
* Example:

powershell

CopyEdit

if (Get-Process -Name "myProcess") {

Write-Host "Process is running"

} else {

Write-Host "Process is not running"

}

**How to Stop (Delete) Processes:**

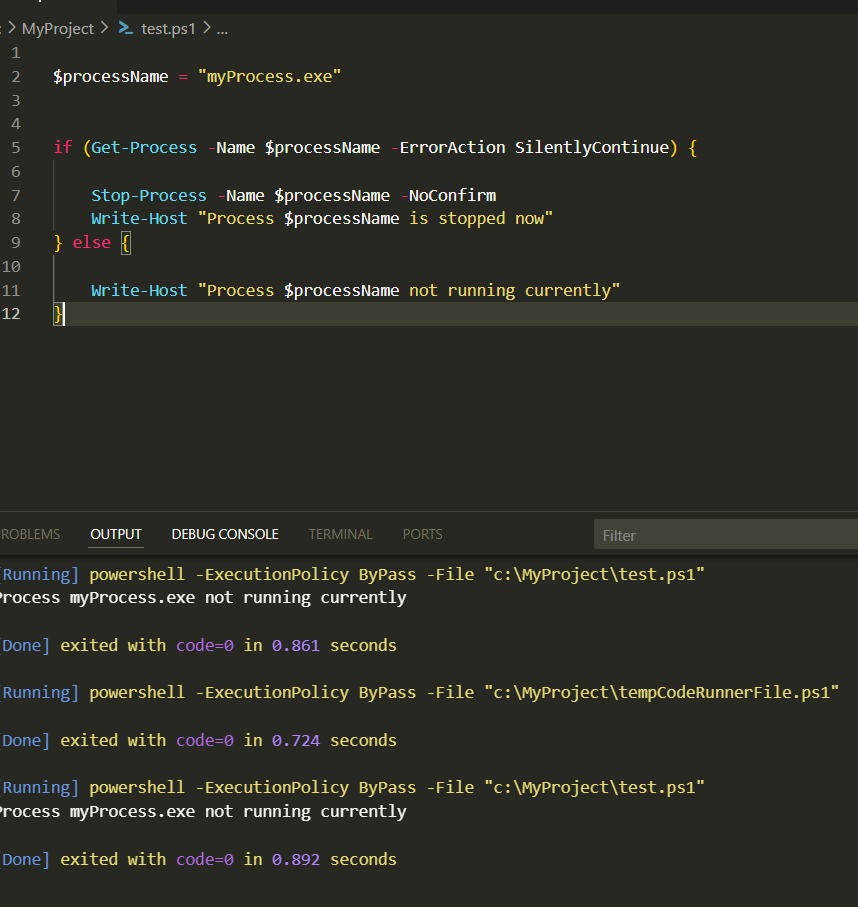
* Use PowerShell’s Stop-Process cmdlet to terminate running processes by name or ID.
* Example:

powershell

CopyEdit

Stop-Process -Name "myProcess" -NoConfirm

* The -NoConfirm parameter suppresses the confirmation prompt.
* Stopping processes requires appropriate permissions and should be done carefully to avoid unintended issues.



**###For Loop in powershell psadt**

**$fruits = @("Apple", "Banana", "Cherry")**

**for ($i = 0; $i -lt $fruits.Count; $i++) {**

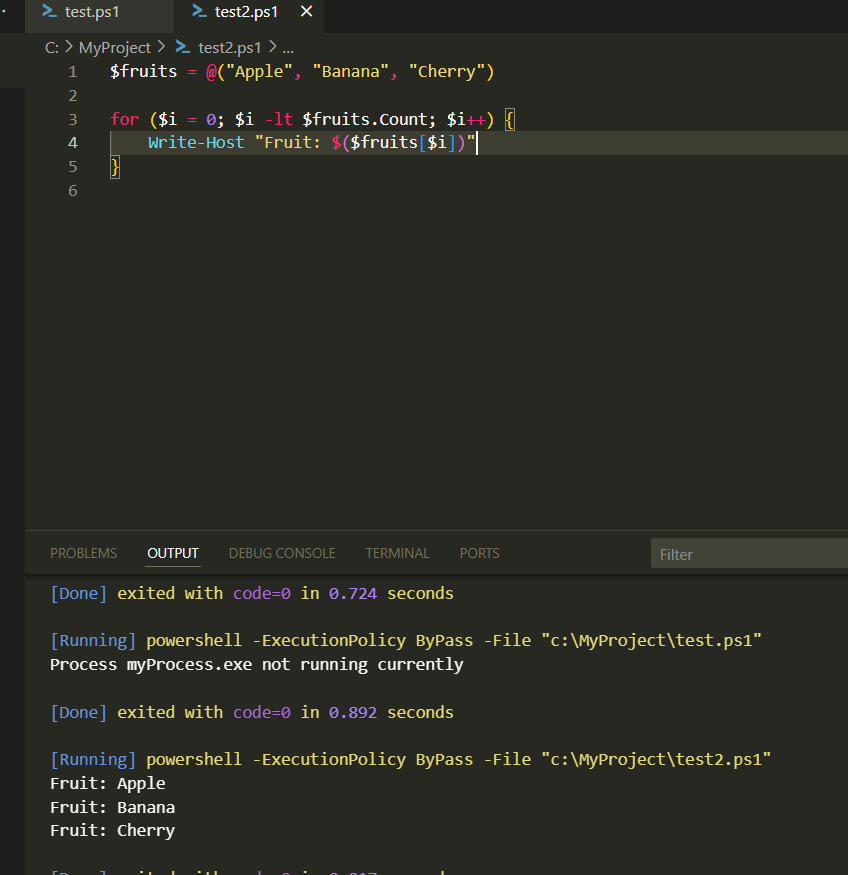
**Write-Host "Fruit: $($fruits[$i])"**

**}**

**This PowerShell script creates an array named $fruits containing three items: Apple, Banana, and Cherry.**

**Then, using a for loop:**

* **A counter variable $i starts at 0.**
* **The loop runs as long as $i is less than the number of items in the $fruits array ($fruits.Count).**
* **In each iteration, Write-Host prints the fruit at the current index $i to the screen.**

****

**Topic 2: Set-ADTIniValue**

1. **Purpose of .ini files**
   * **.ini files store configuration data in sections and key-value pairs.**
   * **Example:**

**InstallPath=C:\Program Files\MyApp**

**Version=1.0**

1. **Creating an .ini file in PowerShell**
   * **You can define content as a string and save it with .ini extension:**

**powershell**

**CopyEdit**

**$iniPath = "C:\MyFolder\config.ini"**

**$content = @"**

**[Settings]**

**InstallPath=C:\Program Files\MyApp**

**Version=1.0**

**"@**

**New-Item -Path (Split-Path $iniPath) -ItemType Directory -Force | Out-Null**

**$content | Out-File -FilePath $iniPath -Encoding ASCII**

* + **This creates the file and its folder if they don’t exist.**

1. **Finding an .ini file**
   * **File Explorer: Search for config.ini and check its Location in Properties.**
   * **PowerShell:**

**powershell**

**CopyEdit**

**Get-ChildItem -Path C:\ -Filter config.ini -Recurse -ErrorAction SilentlyContinue**

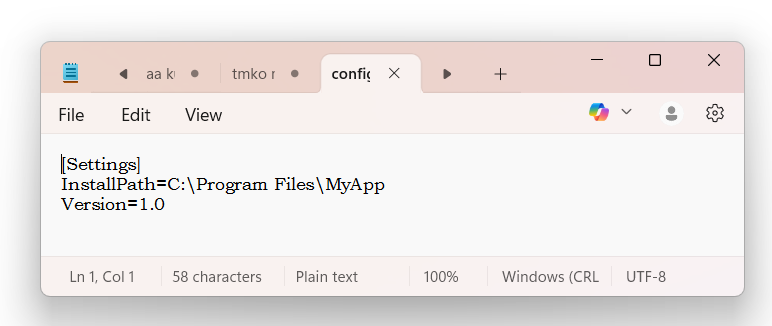
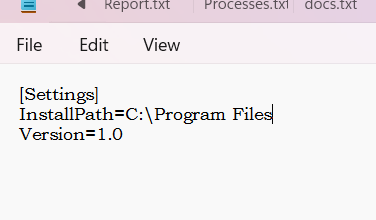
1. **Modifying .ini values with PSADT**
   * **Set-ADTIniValue updates a specific key in an .ini file.**

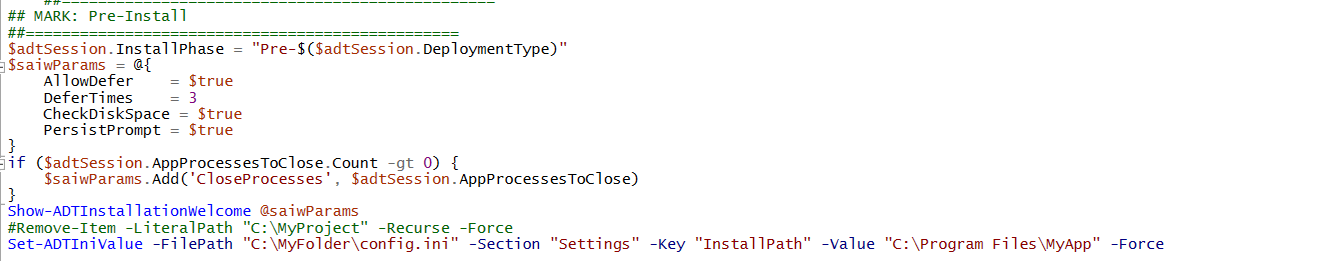
**powershell**

**CopyEdit**

**Set-ADTIniValue -FilePath "C:\MyFolder\config.ini" -Section "Settings" -Key "Installpath" -" C:\Program Files\MyApp ” -Force**

* + **-Force ensures the value is written even if it exists.**

****

****

**Topic 3:**  **Individual Command Logging in PSADT**

**PSADT provides strong logging features to track every step of a deployment script. The Write-Log function is the main way to log custom messages, command executions, and status updates directly into the toolkit’s main log file.**

**Logging Methods:**

**Write-Log – Add custom log entries for messages, commands, or status checks.**

**powershell**

**Copy**

**Edit**

**Write-Log "Starting installation"**

**Execute-Process -Path "msiexec.exe" -Parameters "/i app.msi /qn"**

**Write-Log "MSI command executed."**

**Separate Log for Specific Commands – Some PSADT functions (e.g., Execute-MSI) allow a -LogName parameter to create a dedicated log for that operation.**

**powershell**

**Copy**

**Edit**

**Execute-MSI -Action Install -Path "App.msi" -LogName "AppInstall.log"**

**PowerShell Built-in Logging – For non-PSADT commands, use Start-Transcript and Stop-Transcript to create separate transcript files.**

**Benefits:**

**Clear execution tracking.**

**Easier troubleshooting.**

**Proof of installation results.**

**In short, using Write-Log throughout your PSADT script creates a detailed and easy-to-read record of the deployment process.**

**Topic 4: Catching Return Codes in PowerShell**

In PowerShell, you can capture and handle return codes in three main ways:

Using $LASTEXITCODE

Captures the exit code from the last external program (e.g., .exe, batch file).

0 = success, non-zero = error/failure.

Example:

powershell

Copy

Edit

Start-Process "my\_app.exe" -Wait

$code = $LASTEXITCODE

if ($code -ne 0) { Write-Host "Failed with exit code $code" }

Using try...catch

Best for handling PowerShell cmdlet errors.

Works with terminating errors; set $ErrorActionPreference = "Stop" to treat non-terminating errors as terminating.

Example:

powershell

Copy

Edit

$ErrorActionPreference = "Stop"

try { Get-Content "missing.txt" }

catch { Write-Host "Error: $($\_.Exception.Message)" }

Returning Custom Exit Codes

Use exit <number> to return a specific code from the script.

Commonly used to indicate success/failure for automation tools.

Example:

powershell

Copy

Edit

if ($hasError) { exit 1 } else { exit 0 }

Key Points:

$LASTEXITCODE → external commands only.

try...catch → PowerShell error handling.

$ErrorActionPreference controls how errors are treated.

exit lets you set your own return code.

**Topic 5:** **MSI/MSP Logging in psadt**

In PSADT, MSI and MSP logging is handled via the Execute-MSI and Execute-MSP functions, which simplify deployment, logging, and error handling.

Key Points:

Logging Parameters

-LogName → Sets the log file name (.log auto-appended if missing).

-LogPath → Sets custom log storage path (defaults to PSADT log directory).

-LogVerbosity → Controls detail level (Verbose, Info, Error).

Examples

Install MSI → Execute-MSI -Action Install -Path "C:\App.msi" -LogName "InstallLog" -LogVerbosity Verbose

Apply MSP Patch → Execute-MSI -Action Patch -Path "C:\Patch.msp" -LogName "PatchLog"

Uninstall MSI → Execute-MSI -Action Uninstall -Path "[PRODUCTCODE]" -LogName "UninstallLog"

Custom Log Location

Set $configToolkitLogDir in AppDeployToolkitConfig.xml.

Or configure it early in Deploy-Application.ps1.

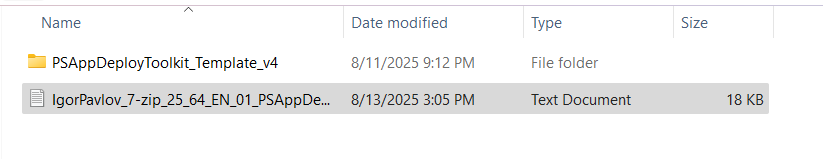
Considerations

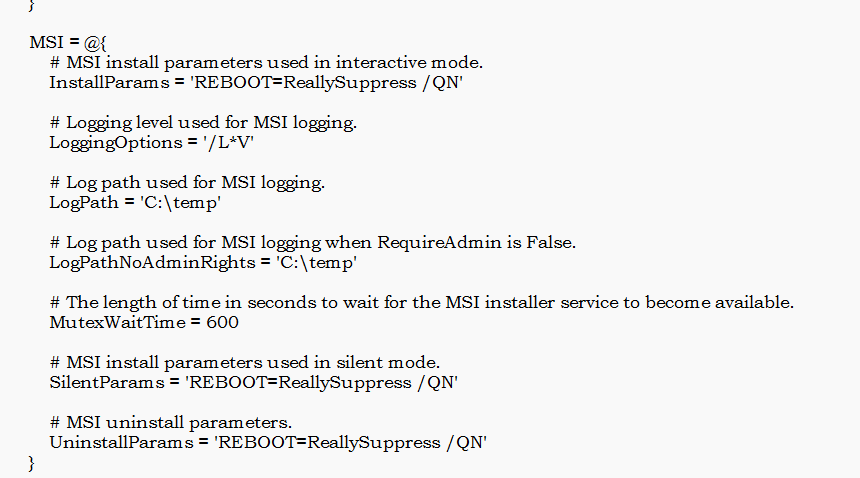
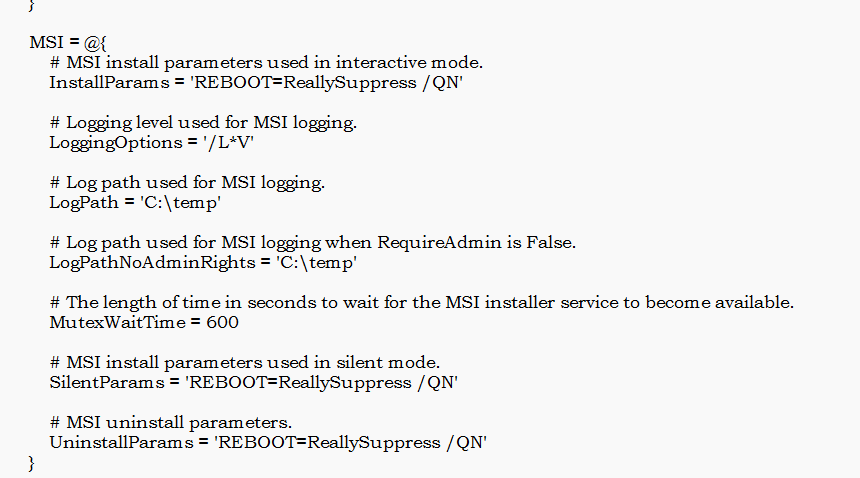
Default location is PSADT working dir — change for network or central logging.

Use -LogVerbosity Verbose for detailed troubleshooting.

Combine with -ContinueOnError for controlled error handling.

This setup ensures complete logging for MSI/MSP installs, patches, and removals, making troubleshooting easier.





**Topic 6:Exception Handling in powershell**

 **Purpose** – try and catch blocks handle errors gracefully without stopping script execution abruptly.

 **try block** – Contains the code that might throw an error (e.g., file operations, MSI install).

 **catch block** – Runs if an error occurs inside try; can log the error, show a message, or take corrective action.

 **finally block** *(optional)* – Executes regardless of success or failure (good for cleanup tasks).

 **PSADT usage** – Often wraps deployment functions (e.g., Execute-MSI) in try/catch to:

* Capture and log errors in the PSADT log file.
* Control whether the script continues or exits on failure.

 **Example**:

powershell

CopyEdit

try {

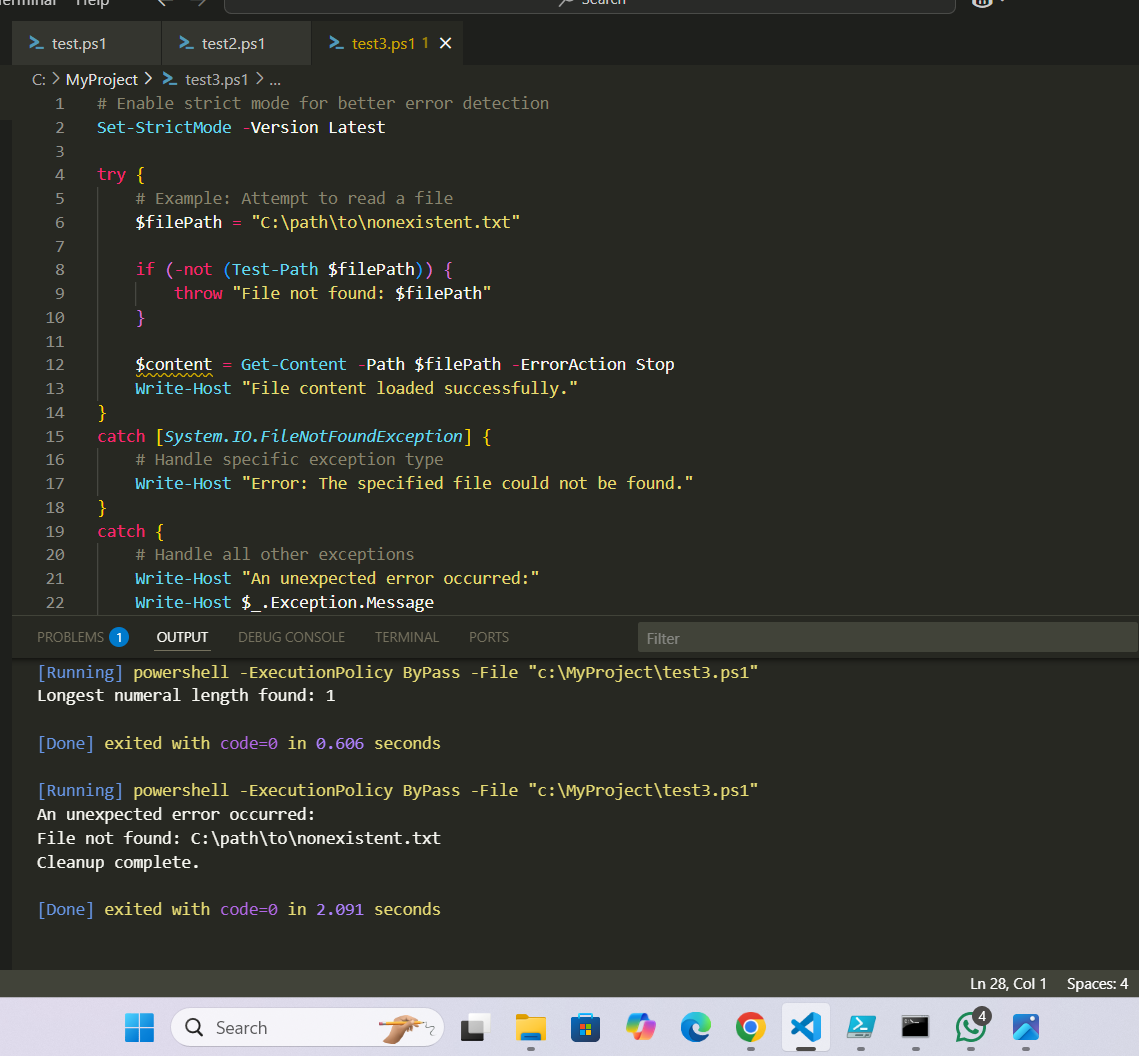
Execute-MSI -Action Install -Path "app.msi"

}

catch {

Write-Log -Message "Installation failed: $\_" -Severity 3

}

 **Benefit** – Prevents silent failures, improves troubleshooting via logs, and ensures smooth deployment flow.

**Topic 7:** **MSIX Overview**

MSIX is a modern Windows app packaging format (improved AppX) that supports both traditional desktop apps (Win32) and modern Store apps.

Combines benefits of MSI, AppX, App-V, ClickOnce with new features like containerization and efficient updates.

Architecture & Components

Package Structure – MSIX is like a zip file containing:

Application files

Configuration files (XML)

Icons, manifest, and optional resources

Containerization – Apps run in isolated containers for better security and stability.

Deployment Methods – Install via Intune, MECM, Microsoft Store, or directly.

Key Features & Benefits

Enhanced Security – Packages are signed and validated.

Efficient Updates – Saves bandwidth and time by updating only changed parts.

Clean Install/Uninstall – Leaves no leftover files or registry junk.

Supports All App Types – Win32, UWP, Store apps.

App Attach – Allows detaching and streaming apps without full installation.

Compatibility – Works with existing Windows deployment tools.

MSIX vs MSI

Security – MSIX is safer with signing/validation.

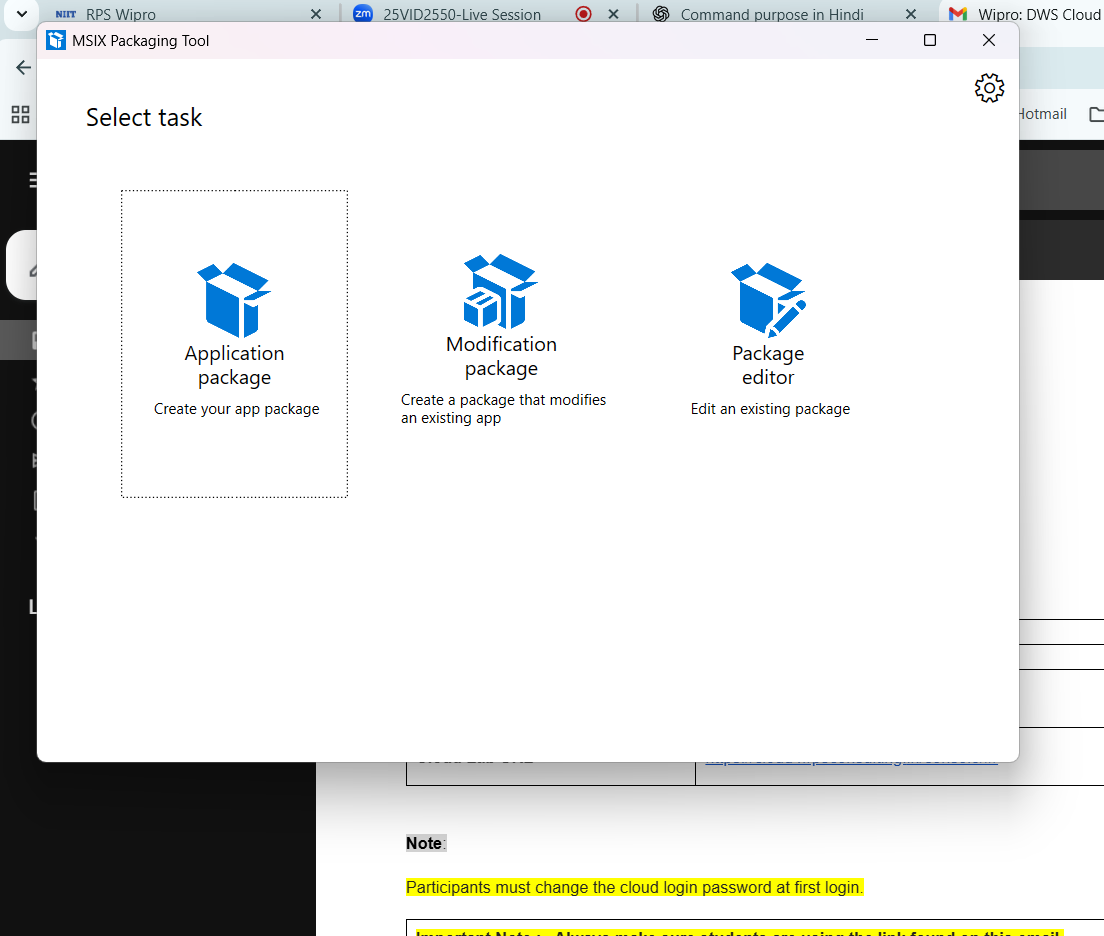
Deployment – Faster and more streamlined.

Updates – More efficient, differential updates.

Isolation – Runs apps in containers for extra protection.

In short:

MSIX is the modern, secure, and efficient way to package and deploy Windows apps, replacing older installers with a container-based, update-friendly approach.



**Topic 8:** **MSIX Benefits and Why MSIX**

MSIX is a modern Windows app packaging format that combines the benefits of older formats like MSI, App-V, and ClickOnce. It improves **reliability, deployment, and security** of applications.

**Key Benefits:**

* **High Reliability:** 99.96% installation success rate with clean uninstall.
* **Reduced Network Usage:** Downloads only necessary data blocks.
* **Disk Space Optimization:** Prevents duplicate files and saves storage.
* **Easy Updates:** Update apps without uninstalling.
* **Security:** Runs in an app container, keeping the OS secure.
* **Simple Deployment:** Supports auto-updates, remote installs, and CI/CD pipelines.
* **Compatibility:** Works with older apps and supports containerization.
* **Less Repackaging Needed:** Developers can package legacy apps directly.

**Why MSIX?**

* **Future-Ready:** Compatible with upcoming Windows updates.
* **Solves Old Problems:** Eliminates file duplication and complex updates.
* **Better User Experience:** Faster installation, smoother updates, and low resource usage.
* **Central Management:** Manage via tools like Group Policy.