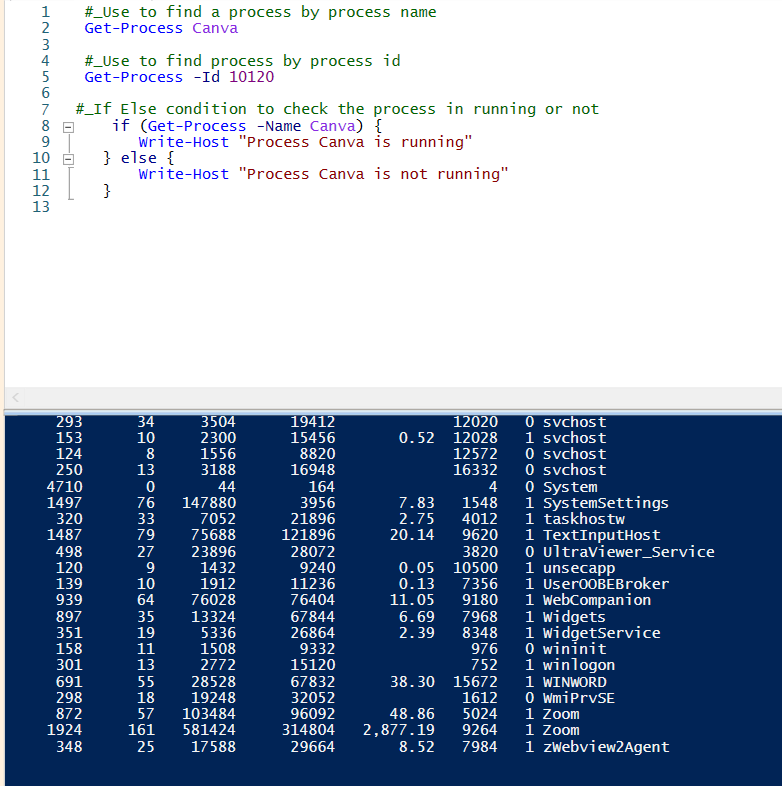
# **PROCESS CHECK & DELETE (Stop process flow) — Windows PowerShell**

Purpose: detect running process(es) and stop them safely during pre/post installation phases.

Example session:  
Get-Process Photos  
Stop-Process -Id 3616  
Stop-Process -Id 22176  
Get-Process Photos -> returns error "Cannot find a process with the name 'Photos'"

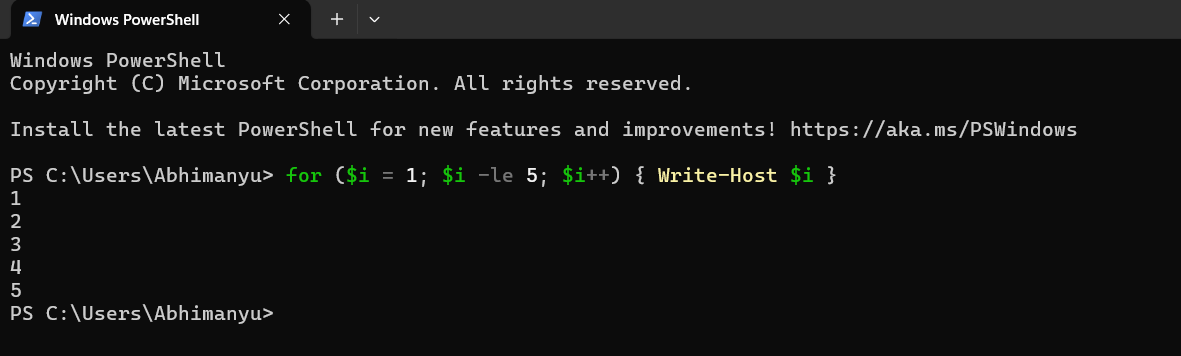
Notes / Best practices:  
• Use Get-Process with either -Id or -Name. Get-Process -Name "Photos" will error if not found — handle that.  
• Always wrap Stop-Process in error handling or check for existence first:

* If we know the Ids: verify they exist before Stop-Process.
* If using names, prefer: $p = Get-Process -Name Photos -ErrorActionSilentlyContinue; if ($p) { $p | Stop-Process -Force }  
  • Running Stop-Process may require administrative privileges; ensure the install session runs elevated.  
  • The sample log I provided shows the expected behavior when processes terminate and later cannot be found — that’s normal. Treat it as informational, not fatal, unless my workflow requires the process be present.



# **LOOP-BASED SCRIPTING OPERATIONS — for-loop examples:**

for ($i = 1; $i -le 5; $i++) { Write-Host $i } Explanation:  
• Initialization: $i = 1.  
• Condition: **$i -le 5** → Checks if $i is **less than or equal to** 5.  
• Increment: **$i++** → Increases $i by **1** each time the loop finishes an iteration.  
• Body: Write-Host prints current values.



# **SET-INI VALUE — Aura.ini example**

What this does:  
This command uses the toolkit helper Set-ADTIniValue to update the Aura.ini configuration file. It targets the [Lighting] section and sets the Brightness key to 100. The command is intended to be executed in the same session or environment where the toolkit helpers are available.

Prerequisites (must verify before running):

1. The toolkit function Set-ADTIniValue must be available in your PowerShell session (run inside the toolkit context or dot-source the toolkit module).
2. You must have write permission to C:\Program Files\ASUS ARMOURY CRATE Service AuraPlugin\Aura.ini (run PowerShell elevated as Administrator).
3. Confirm the file exists and is the correct target file for your changes.

A screenshot of a computer program

AI-generated content may be incorrect.

# **TOOLKIT LOGGING LOCATION & HOW IT’S CONFIGURED (ps1 initialized through xml)**

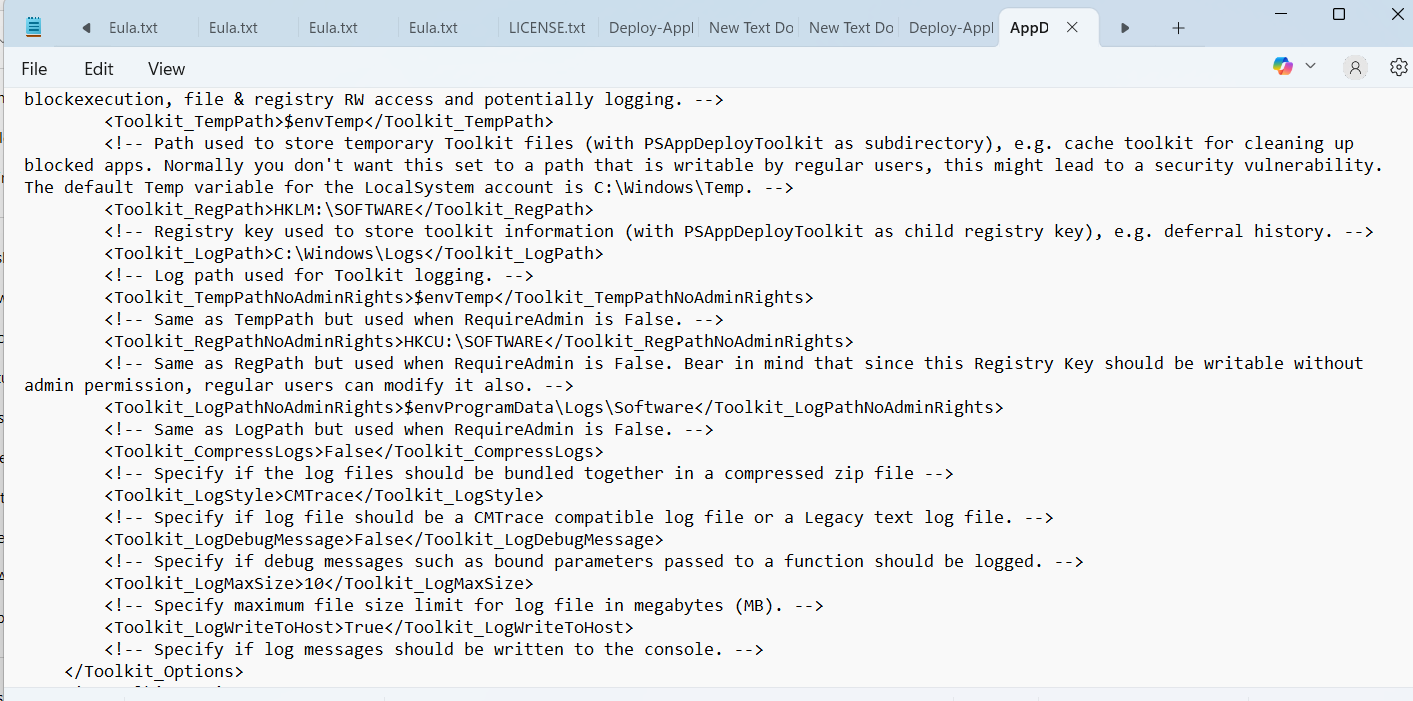
• My Deploy-Application wrapper reads the AppDeployToolkitConfig.xml at startup and builds log paths from Toolkit\_LogPath and MSI\_LogPath.

• Ensure the configured folder exists or the toolkit creates it. If not, create it in the preinstall phase: New-Item -Path $Toolkit\_LogPath -ItemType Directory -Force.

• Keep log path values absolute and avoid user-temp if you need persistent logs for debugging.

Practical checklist:

1. Confirm XML is readable by the account running the installer.
2. Confirm Deploy-Application/Deploy-Toolkit code actually consumes <Toolkit\_LogPath> and <MSI\_LogPath> — search for those tags inside AppDeployToolkitMain.ps1.
3. Ensure $DisableLogging switch behavior is tested (see section 6).



# **MSI / MSP LOGGING — how it’s handled in my script**

Relevant fragment from my script where MSI is executed:  
Execute-MSI -Action 'Install' -Path '7z2501-x64.msi' -Parameters '/QN'

• Execute-MSI is the AppDeployToolkit helper which should direct MSI logging to the MSI\_LogPath configured in the XML. Confirm the toolkit’s Execute-MSI implementation includes /l\*v "<path>\msilog.txt" or uses logging switches appropriate for msiexec.

• When using /QN (quiet) you will not see UI; always preserve logs for troubleshooting. Example msiexec pattern (if toolkit doesn’t already do it):  
msiexec.exe /i "7z2501-x64.msi" /qn /l\*v "C:\Users\adity\Desktop\log\7zip\_install.log"

# **INDIVIDUAL COMMAND LOGGING WITHIN SCRIPT (My ps1 style)**

From my script: the toolkit module is dot-sourced and I use Write-Log and Show-InstallationPrompt for user feedback. I also set $DisableLogging capability.

Key points:  
• The App Deploy Toolkit typically exposes Write-Log -Message <msg> -Severity <level>. Use that for every meaningful action (pre-install checks, copies, service stops). That ensures commands appear in the toolkit-log and you’ll have a full trace.

• Use Try { <command>; Write-Log "Command succeeded" } Catch { Write-Log "Error: $\_" -Severity 3 }. This keeps every command traceable.

• Avoid Write-Host for recorded logs; Write-Host is ephemeral and not automatically captured by log files.

Concrete example pattern:  
Try {  
Execute-MSI -Action 'Install' -Path '7z2501-x64.msi' -Parameters '/QN'  
Write-Log -Message "MSI install completed for 7z2501-x64.msi" -Severity 1  
}  
Catch {  
Write-Log -Message "MSI install failed: $($\_.Exception.Message)" -Severity 3  
$mainExitCode = 1  
}

# **HOW TO CATCH RETURN CODES IN POWERSHELL — using my script context**

My script already uses $mainExitCode, Exit-Script -ExitCode $mainExitCode, and Try/Catch around critical sections. That pattern is correct — make it consistent.

Practical rules:  
• For external executables, check $LASTEXITCODE immediately after the call. Example:  
& "msiexec.exe" /i "7z2501-x64.msi" /qn  
$rc = $LASTEXITCODE  
If ($rc -ne 0) { Write-Log -Message "msiexec failed with $rc" -Severity 3; $mainExitCode = $rc; Exit-Script -ExitCode $mainExitCode }

• For toolkit helper functions (Execute-MSI, Execute-Process), those functions often throw on failure or return objects — consult their design and handle accordingly:  
Try { Execute-MSI -Action 'Install' -Path $msiPath -Parameters '/QN' }  
Catch { $mainExitCode = 60002; Write-Log -Message "Execute-MSI failed: $($\_.Exception.Message)" -Severity 3; Exit-Script -ExitCode $mainExitCode }

• Preserve meaningful exit codes for SCCM (your script already maps errors to specific numeric codes like 60001 etc.)