ChatGPT

PowerShell Automation Plan for HP CMSL & HPIA in Tanium

Overview

This document outlines a **modular PowerShell-based solution** to automate HP Client Management Script Library (CMSL) and HP Image Assistant (HPIA) for firmware and driver packaging in a Tanium environment. The solution is divided into distinct components (scripts and configurations) that handle CMSL installation, repository management for HP driver/firmware updates, package building for Tanium deployment, logging, and documentation. Each component is detailed below with implementation steps, best practices, and recommendations.

The goal is to streamline the process of maintaining up-to-date HP SoftPaq repositories for multiple hardware platforms (models) and packaging them (with HPIA and install scripts) for deployment via Tanium. By using a centralized JSON configuration and modular scripts, the system will be maintainable and extensible. Key considerations such as silent installation of tools, offline repository sync, conditional downloading of new versions, robust logging, and security checks are included.

CMSL Installation

The first module handles the **installation of HP Client Management Script Library (HPCMSL)** on the packaging server. This ensures all CMSL PowerShell cmdlets are available for subsequent steps. Implementation details:

- **Automated Download**: The script will download the latest CMSL installer (EXE) from HP's official source. For example, version 1.8.2 is available at the URL: `https://hpia.hpcloud.hp.com/downloads/cmsl/hp-cmsl-1.8.2.exe`.
  The script should be able to update this URL if a newer version is required in the future (e.g., by storing the URL or version in the config file).
- **Version Check**: Before downloading, the script checks if CMSL is already installed and its version. This can be done by trying to import the `HPCMSL` module or checking installed programs. If the required version (or a newer one) is present, skip download; otherwise proceed to download the EXE.
- **Silent Installation**: Install the CMSL EXE with silent flags to avoid user intervention. The installer supports standard Inno Setup switches: `/VERYSILENT /NORESTART /SUPPRESSMSGBOXES` [1].
  These options ensure the installation runs quietly, does not reboot the system, and suppresses any pop-up dialogs. (If needed, include `/SP-` to skip the initial prompt as well [2], although `/VERYSILENT` usually implies that.)
- **Execution**: Use PowerShell's `Start-Process` or `Invoke-Expression` to run the installer with the above arguments. For example:

```
$installer = "$env:TEMP\hp-cmsl-1.8.2.exe"
Invoke-WebRequest -Uri $cmslUrl -OutFile $installer
Start-Process -FilePath $installer -ArgumentList "/VERYSILENT", "/
NORESTART", "/SUPPRESSMSGBOXES" -Wait
```

This will wait for installation to complete. Afterwards, the HPCMSL PowerShell modules should be available on the system (typically it installs modules under Program Files).

- **Validation**: After install, verify that the `Import-Module HPCMSL` works and the module version matches expected (e.g., 1.8.2). Log success or error accordingly. If installation fails or module commands are not available, the script should raise an error and halt further processing (since CMSL is prerequisite for everything else).

By automating CMSL setup, the solution ensures the correct environment for repository management. Running this step with administrative privileges is required (as installing software). It's a best practice to validate the installer's authenticity – for instance, check the digital signature of the downloaded EXE or its hash against HP's published checksums (if provided) to ensure integrity and security before execution.

Repository Management

This section outlines how to manage **offline SoftPaq repositories** for each supported HP platform (model) using CMSL. The aim is to maintain a structured repository of drivers/firmware for each model and OS, filtered to include only relevant updates, which HPIA can use offline. The script responsible for this will handle repository initialization, syncing (updating), and applying the central configuration filters.

Repository Initialization & Structure:
For each platform defined in the config (see Config File section), the script will ensure a repository folder exists: - Construct the repository path. This can be a base path (e.g., `D:\HPRepos`) plus an optional OS directory and platform identifier. For example, one scheme could be `D:\HPRepos\Win11\Platform_87EF\` for platform ID 87EF on Windows 11, or simply `D:\HPRepos\87EF\` if not segregating by OS. - If the directory is missing or not yet a repository, **initialize it** with CMSL:

```
Set-Location -Path "D:\HPRepos\Win11\Platform_87EF"
Initialize-Repository
```

This creates the necessary `.repository` metadata subfolder [3]. Initialization is done once per new repo. If the repo already exists (i.e., the folder has a `.repository`), skip this step to preserve existing data (unless a rebuild is forced). - After initialization, **add repository filters** appropriate for that platform if not already set. The filters define what content to include. Using `Add-RepositoryFilter` cmdlet, specify: - **Platform ID**: the 4-digit hex code for the model (from HP's platform list). This is mandatory [4]. - **OS and Version**: target OS (e.g., `win10` or `win11`) and OS release (e.g., `"22H2"` for Windows 10/11 22H2) [5] [6]. These should come from the config. Explicitly setting these ensures the repo only pulls SoftPaqs for that OS version (if not specified, CMSL assumes current OS, which might not be intended [5]). - **Categories**: filter to only certain SoftPaq categories as needed. For driver/firmware packaging, likely categories such as *Driver, BIOS, Firmware* (and possibly *Software* or *Utility* if desired) are used. If not specified, all categories are included, but we will usually restrict to avoid irrelevant packages [7]. The central config can list categories

to include or exclude. For instance, one might include `Driver` and `Firmware` but exclude things like `Software` or `Diagnostic` to keep the package lean. - **Release Types**: optionally filter by importance (Critical, Recommended, etc.) [8]. For example, include Critical and Recommended updates, skip Routine if not necessary. - **Other filters**: Characteristics (SSM, etc.) if needed, or `-PreferLTSC` if using LTSC-specific reference files [9]. These can be driven by config flags. - The script should check existing filters (using `Get-RepositoryInfo` or `Get-RepositoryConfiguration`) to decide if filters need to be added or updated. If the repository already has the correct filter set for that platform/OS, avoid adding duplicates. If filters differ from config (e.g., categories changed), update them by removing old filters and adding new ones accordingly. - Enable **Offline Mode** for HPIA: Configure the repository so that it contains all auxiliary files needed by HPIA during offline analysis. This is done via:

```
Set-RepositoryConfiguration -Setting OfflineCacheMode -CacheValue Enable
```

Enabling OfflineCacheMode ensures that platform advisory files, BIOS keys, etc., are downloaded too [10] [11], making the repository self-sufficient for HPIA's use. We only need to set this once per repository (it's a persistent setting stored in `.repository` config). Additionally, set error handling to continue on missing files:

```
Set-RepositoryConfiguration -Setting OnRemoteFileNotFound -Value LogAndContinue
```

This way, if a particular SoftPaq reference is broken or missing on HP's servers, the sync won't abort the entire process [12] [13] – it will log the issue and proceed.

Repository Sync & Update:
With the repository initialized and filters configured, the script performs a sync to download or update the SoftPaq files: - Invoke the synchronization using `Invoke-RepositorySync`. This will connect to HP's repositories and download any new or updated SoftPaqs that match the filters for that model and OS [14] [15]. By design, `Invoke-RepositorySync` only downloads the **latest versions** of SoftPaqs for each filter criterion (older superseded versions remain until cleanup is run). The process also updates metadata like the SoftPaq index (.CVA files) each time [16]. - The script should capture the output or result of the sync. If run without `-Quiet`, it provides progress which can be logged. If any errors occur (e.g., network issues or missing SoftPaqs), these should be logged and possibly emailed if alerting is configured (CMSL supports adding sync failure email recipients via `Add-RepositorySyncFailureRecipient`, though that may be optional). - After sync, run `Invoke-RepositoryCleanup` to remove obsolete SoftPaq files from the repository [17]. This will delete any older versions that have been superseded by newer downloads, keeping the repo streamlined to latest drivers only. The cleanup helps manage disk usage and ensures only current updates are packaged. - **Conditional Sync**: The script can support a "dry run" or "check mode" (triggered by a flag) where it only checks if updates are available (maybe via `-WhatIf` or using CMSL's report generation) without actually downloading, and logs what would change. In normal runs, it will perform the actual sync. We also incorporate a **"force sync"** option (through a command-line switch) that disregards any cached results and forces a full re-scan/download – useful if we suspect the repository might have missed something or after a long downtime.

HPIA Download Management:
HPIA (HP Image Assistant) is a separate utility needed on the endpoint for installing the updates from the

repository. We maintain a local copy of the HPIA executable and update it only when necessary: - Decide on a storage for HPIA binary (for example, a folder like `D:\HPRepos\HPIA\hp-image-assistant.exe` or include it in each repository folder). A single copy is typically sufficient, which can then be included in each package. - **Check current version**: The script should determine the current HPIA version available locally. If none is present, or if an update check indicates a newer version is available, it proceeds to download. - **Determine latest version**: To avoid downloading HPIA every run, implement a check. For example, the script could query an HP source for the latest version number. HP's site provides the latest version and SoftPaq ID for HPIA (e.g., version 5.3.2 corresponds to SoftPaq `sp158107` released 2025-04-25 [18] [19]). The script might fetch a known URL (like an HP API or parse the HPIA HTML page) to get the latest version string. Alternatively, maintain the expected latest version in the config and update it manually when informed of new releases. - **Download if needed**: If the latest version is newer than the local copy, download from the official source. HPIA is available on the same HP cloud (`hpia.hpcloud.hp.com`) in the format `hp-hpia-<version>.exe` [20]. For example, for 5.3.2 the URL is `https://hpia.hpcloud.hp.com/downloads/hpia/hp-hpia-5.3.2.exe`. Use `Invoke-WebRequest` to download it to the designated path. - **Verify integrity**: As a security best practice, verify the HPIA EXE after download. This could involve checking the digital signature (it should be signed by HP Inc.) or computing a hash and comparing it to a known value if HP publishes those. Only proceed if verification passes, otherwise log an error and do not use the new file. - The script should **not re-download HPIA if the local version is already current**. It can log a message like "HPIA is up to date (v5.3.2), download skipped." This saves bandwidth and time. - If HPIA was updated, consider copying the new HPIA EXE into each repository folder or prepare to include it during packaging (depending on packaging approach). We likely will include one copy per package for simplicity.

Multi-Platform Support:
The repository management logic runs for **multiple platforms/models** in one execution. The central config will list each platform (identified by platform ID and model name) that we support. The script will loop through each entry and perform the above steps. Key considerations: - Isolation: Each model has its own repository directory (to avoid mixing drivers between different models). This is important because we will package each model's repository separately. The developer blog confirms using separate folders per model for HPIA repositories [21]. - Efficiency: If needed, the script can perform syncs sequentially. Optionally, some tasks could be parallelized (PowerShell supports jobs or parallel foreach) to update multiple model repositories concurrently, though caution with bandwidth and system load should be considered. In most cases, sequential is simpler and the runtime is manageable if only a moderate number of models are supported. - Config-driven filtering: Each platform could have different OS or category filters. For example, newer models might be on Windows 11 only, older on Windows 10. The config can reflect that and the script will apply accordingly for each repository. - Post-sync, the script can generate a summary report (perhaps by using `Get-RepositoryInfo` or custom logic) of how many SoftPaqs each repo contains, last update time, etc., and include that in logs or output. This helps in tracking the content of each repository over time.

By the end of repository management, we have an up-to-date set of folders, each containing the latest drivers/firmware (SoftPaqs) for a given HP model and OS, with an offline-capable configuration that HPIA can use. These repositories are now ready to be bundled into deployment packages.

Config File

A centralized configuration file governs the behavior of the scripts and defines which platforms to support. We use a **JSON format** for this config so it's human-editable and machine-readable. However, JSON doesn't

natively support comments, which are useful for documentation in config files. To address this, we use **JSON with comments** (sometimes called JSON5 or JSONC) or implement a workaround: - **JSON with Comments**: In PowerShell 6 and above, `ConvertFrom-Json` can parse JSON even if it contains comments (they will be ignored) [22]. This means we can include lines starting with `//` (single-line comments) or `/* ... */` (block comments) in the config file to explain settings, and the parser will still produce the object (comments are dropped from the result). This approach requires running the script with PowerShell Core/7+, which is recommended for modern environments. If the environment is stuck on Windows PowerShell 5.1, an alternate approach is needed (since PS5.1 errors on comments). - **Alternate Workaround**: For PS5.1 compatibility, we could implement a preprocessing step: read the config file as text and strip out comment lines before using `ConvertFrom-Json`. Another approach is to use a JSON5 parser library or module if available. But a simple regex to remove `//` comments and block comments would suffice for our needs. This way, we can still allow the user to annotate the JSON file with notes.

Config File Structure:
The JSON config defines: - **Supported Platforms**: An array of objects, each with details for one HP platform (model). Fields likely include: - `PlatformID` – The HP platform ID (hex string) for the model [4]. - `Model` – Human-readable model name (optional, for clarity or logging). - `TargetOS` – Which OS this platform's repo is for (e.g., `"win10"` or `"win11"`). - `TargetOSVersion` – The OS version or release (e.g., `"21H2"` or `"22H2"`). - `Categories` – List of categories to include (e.g., `["Driver","Firmware","BIOS"]`). Could also support an "All" or leave empty to include all [7]. - `ReleaseTypes` – (Optional) list of release types to include (e.g., `["Critical","Recommended"]`). If omitted, all types are included [8]. - Other platform-specific settings if needed (like `PreferLTSC` flag, or maybe a custom filter). - **Global Options**: Settings that apply to the overall operation. For example: - `RepositoryBasePath` – the root directory where repositories are stored (e.g., `D:\\HPRepos`). - `OrganizeByOS` – boolean or pattern indicating if repos are under OS-specific subfolders. If true, the script will create/use subdirectories per OS version (like `...\Win10\...` vs `...\Win11\...`). Alternatively, this could be encapsulated in the path template itself. - `VerboseLogging` – default verbosity setting (can be overridden by command-line switches). - `DryRun` – a flag to indicate if by default the script should simulate (useful for testing). - `ForceRebuildPackages` – default behavior for whether to rebuild packages even if no changes (this might tie into the packaging step). - `HPIA_Version` – optionally, store the expected HPIA version or the last downloaded version to compare against updates. - `Use7zip` – whether to prefer 7-Zip for compression if available. - etc., as needed (like log file path, retention settings, etc.). - **Behavior Flags**: Could include toggles like `DisableCleanup` (if one wants to skip deleting old SoftPaqs for some reason), or `EmailOnError` if we integrate email notifications for failures, etc. These can be expanded in the future.

Using JSON (with possible comments) allows the config to be easily extended. Below is an illustrative snippet of how the config might look (comments included for clarity):

```
{
  // Base path for all repositories
  "RepositoryBase": "D:\\HPRepos",
  // Whether to organize repos by OS (creates subfolders per OS)
  "OrganizeByOS": true,
  // Global default flags
  "VerboseLogging": false,
  "DryRun": false,
```

```
  "ForceRebuildPackages": false,
  // Define platforms to support
  "Platforms": [
    {
      "PlatformID": "87EF",  // Example platform ID (hex)
      "Model": "HP EliteBook 840 G7",
      "TargetOS": "win10",
      "TargetOSVersion": "22H2",
      "Categories": ["Driver","Firmware","BIOS"],
      "ReleaseTypes": ["Critical","Recommended"]
    },
    {
      "PlatformID": "8641",
      "Model": "HP ZBook 15 G6",
      "TargetOS": "win11",
      "TargetOSVersion": "21H2",
      "Categories": ["Driver","Firmware"],
      "ReleaseTypes": ["Critical","Recommended","Routine"]
    }
    // ... more platforms ...
  ]
}
```

In this JSON5 example, comments (like the lines starting with `//`) would be ignored by the script's parser (assuming PS7) [22]. Each platform entry provides the needed inputs for our repository and packaging scripts. The config can be placed in a file like `hp_update_config.jsonc` and should be stored in a known location. Our scripts will load this file at start (`Get-Content | ConvertFrom-Json`) to drive their logic. If any required field is missing (like a PlatformID or OS), the script should throw a clear error indicating a config problem.

Validation: The script should validate the config after parsing: - Ensure `RepositoryBase` exists or can be created. - Ensure each platform object has the required fields (PlatformID, TargetOS, etc.). Perhaps verify PlatformID is a 4-character hex string (maybe even ensure it's valid by CMSL's `Get-HPPlatformData` or similar, but that might be excessive). - Possibly verify that no two platforms have the same output folder path (to avoid conflicts). - If `OrganizeByOS` is true, ensure TargetOS/OSVersion are given for each platform. - Log any config issues and halt execution if critical issues are found.

Having a well-documented config file allows easy addition of new models in the future or adjustment of categories/filters without modifying the scripts. This design makes the solution **extensible** – e.g., supporting a new HP model is as simple as adding an entry in JSON and re-running the sync and packaging.

Tanium Package Building

After repositories are updated, the next step is to **package them for deployment** via Tanium. This section describes a separate script (or function) that takes each platform's repository and creates a deployable archive (ZIP) along with the necessary install script and HPIA.

Packaging Overview: For each platform repository (one per HP model as built above), we will create a ZIP file containing: - All SoftPaq files for that model (the contents of the repository folder, except perhaps the `.repository` metadata folder which might not be needed on the client side). - The HPIA executable (so that the target machine has HPIA to run – including it avoids the endpoint downloading it separately). - A **universal install script** (e.g., a PowerShell script or batch file) that will orchestrate the installation of drivers/firmware on the endpoint using HPIA and the repository content.

Each ZIP will essentially be a self-contained package that Tanium can distribute to endpoints. The packaging script will iterate through each platform's repository directory and perform the following:

Zipping Process: - Determine the output file name and path. A recommended naming convention could be: `<Model>_<OS>_Drivers.zip` or use the platform ID: `HPRepo_<PlatformID>_<OSVer>.zip`. For clarity and uniqueness, including model name and OS version is helpful (e.g., `EliteBook840G7_Win10_22H2.zip`). - Choose the compression method: - **7-Zip (if available)**: Check if the system has `7z.exe` accessible (perhaps the script looks for it in the PATH or a known location). 7-Zip's deflate implementation is faster and can create zip files with better compression ratios. If found, use it to create the zip. For example:

```
$zipOut = "EliteBook840G7_Win10_22H2.zip"
$sourceFolder = "D:\HPRepos\Win10\Platform_87EF"
& 7z.exe a -tzip $zipOut "$sourceFolder\*"
```

This adds all files from the repository folder into a ZIP archive. (We may want to exclude the internal `.repository\` folder with `-x` switch if it's not needed on client. The `.repository` contains logs and config used by CMSL but HPIA offline mode may not require it. However, it might contain the index files that HPIA could use. To be safe, we could include everything; the size impact is minimal.) - **Fallback to Windows compression**: If 7-Zip is not installed, use PowerShell's `Compress-Archive` cmdlet or another method to create the zip. e.g.,

```
$zipOut = "EliteBook840G7_Win10_22H2.zip"
Compress-Archive -Path "$sourceFolder\*" -DestinationPath $zipOut -Force
```

This is simpler but note that `Compress-Archive` can be slower and sometimes struggles with very large files or deep paths. We should ensure the repository paths are not too long (Windows has path length limits for zip items). If path lengths are an issue (SoftPaq names can be long), enabling long path support on the system or using 7-Zip might avoid problems. - Include the **HPIA binary** in the zip: If the HPIA EXE was stored centrally (e.g., `D:\HPRepos\HPIA\hp-hpia-5.3.2.exe`), copy it into the repository folder (or specify it in the compress command) so that it ends up in the archive. It might be placed at the root of the zip for easy access. - Include the **install script**: We will prepare a PowerShell script, say `Install-HPUpdates.ps1`, that is designed to run on the endpoint. This script will be added to each ZIP (possibly at root). The packaging script can ensure the latest version of this install script is copied into each repository folder (or directly added into the zip).

Install Script (Endpoint): This is a crucial piece that runs on the client via Tanium to apply the drivers/firmware: - The script will detect the local machine's model and platform. Using WMI is a straightforward

method: for example, querying `Win32_ComputerSystem` for the Model name, or `Win32_ComputerSystemProduct` for the SKU. For HP specifically, `Win32_ComputerSystemProduct.Version` often contains the product number, and `.Name` might have the family. But simplest is `Get-WmiObject -Class Win32_ComputerSystem | Select-Object Model` which yields a model string (e.g., "HP EliteBook 840 G7"). - **Matching logic**: If each Tanium package is targeted to the correct model, then the script's job is easier (it can assume the included drivers are for this machine). However, to build a universal script, we consider the case where a package might contain multiple model drivers or if the same script is reused: - The script can have a mapping of expected PlatformIDs or model names to the content. For example, it could look at the current directory for any `.zip` or subfolder matching the system's platform ID or model name. If our package only contains one set of drivers, there might only be one candidate. - In a scenario where the package is combined (say we delivered multiple model zips at once), the install script would iterate through available zips to find one that matches this system. The config within the script might list the known platform IDs it supports. If a match is found, proceed with that. If none matches, as a fallback the script could log a warning and choose the first available package as a default. (This fallback ensures that if the model detection fails or if the model wasn't anticipated, it still attempts an installation rather than doing nothing – though the result might be suboptimal if it's the wrong drivers.) - **Extraction**: The script should extract the repository content and HPIA from the zip (if Tanium hasn't already done so). Depending on how Tanium deploys, we might need to explicitly extract: - If Tanium can deploy a package with multiple files, we might not need to zip at all; but typically, Tanium deploys a package as one file and runs commands. We assume we deliver a zip and then run the script to extract it. - Use built-in tools to extract: PowerShell can extract zip via `Expand-Archive`. Or, if we included a small self-extractor or if endpoints have 7-Zip, but it's safer to assume they don't. So likely:

```
Expand-Archive -Path .\EliteBook840G7_Win10_22H2.zip -DestinationPath C:
\HPUpdates -Force
Set-Location C:\HPUpdates
```

Now C:\HPUpdates should contain the repository files, HPIA exe, etc. - **Run HPIA**: With everything extracted, the script will execute HPIA to analyze and apply updates from the offline repository: - We run HPIA in offline mode pointing to the repository. For example:

```
.\hp-hpia-5.3.2.exe /OfflineMode:"C:\HPUpdates" /Operation:Analyze /
Action:Install /Silent
```

The `/OfflineMode:<path>` tells HPIA to use the given folder as the repository for analysis [23]. We combine it with an instruction to automatically install recommended updates. HPIA has an `/Auto` or combined analyze+install switch, which is equivalent to `Operation:Analyze` and `Action:Install` on the local machine [24]. Using these ensures HPIA will scan the system, compare against the repository content, and install any newer drivers, BIOS, or firmware that apply. - We might include additional HPIA parameters: `/Category:Drivers,Firmware` if we want to restrict categories (though our repo itself is filtered, HPIA by default might consider all categories present). There is also a `/Force` switch introduced in newer HPIA which forces installation of all compatible SoftPaqs in the repo regardless of current version [25]. Using `/Force` can ensure even if a driver is same version it might reinstall, but we might not want that normally. We could expose this as a config flag if needed for certain scenarios. - Use `/Silent` or

noninteractive mode so that end users don't see prompts. HPIA's `/NonInteractive` or `/Silent` options will suppress any GUI. - Logging: instruct HPIA to output a log file for its actions (HPIA can produce a CSV or HTML report of what it did). For example, use `/LogFolder` to specify a path for logs or `/AutoReport` to generate a CSV of installed SoftPaqs. - **Post-install**: After HPIA runs, the script can check the exit code or output. HPIA's exit code might indicate if a reboot is required after some updates (for instance, BIOS update might require reboot). If so, the script could signal this by creating a file or a specific exit code that Tanium can capture to know a reboot is needed. Alternatively, it can simply initiate a reboot if policy allows (though typically you'd let the management system handle reboot logic). - Clean up the extracted files (optional): After installation, to conserve disk, the script can delete the extracted repository files from `C:\HPUpdates`. (However, if troubleshooting is needed, one might retain logs and perhaps the repository until success is confirmed. Maybe only clean after a successful run.) - The install script itself should implement logging (writing to console and a file) so that we have records on the endpoint of what happened (particularly important if troubleshooting a deployment on a specific machine).

This **universal install script** is included with each package. For consistency, you might maintain a single script template and the packaging script just copies it in. If any model-specific logic is needed, the script can read the model or platform from WMI and adapt (but ideally avoid per-model branching; the goal is one script fits all, using dynamic detection).

Performance Considerations: - The size of each package could be several gigabytes if including all drivers. Using compression helps; drivers (SoftPaq EXEs) often compress well since they contain lots of binaries. 7-Zip's default compression for .zip is Deflate which should be fine; if more compression is needed, one could consider .7z format, but then the endpoint would need a way to extract it (Tanium might not handle .7z natively). Likely we stick to standard .zip for compatibility. - Building the zip: We should try to do it efficiently. If using 7-Zip, the process is quite fast. If using `Compress-Archive`, note that it loads everything into memory; for very large sets, it might be slow or memory-intensive. In such cases, splitting the workload or ensuring adequate memory is important. Another option is using the `[System.IO.Compression.ZipFile]` .NET class directly with streaming – but 7-Zip is simpler. - If the repository has many small files, compression overhead might be significant; but since SoftPaqs are typically delivered as single EXEs per driver, we mostly deal with fewer large files.

Tanium Integration: - The script could optionally automate the creation or update of Tanium package definitions via Tanium's API or CLI, but that is beyond our current scope. The assumption here is that an administrator will take the produced zip and install script and create a Tanium package (or update an existing one) with them. The README (see Documentation) will provide instructions for that. - If automation is desired: after building each zip, the script could call Tanium's API to upload the file and set up a package that executes the PowerShell install script. This would require Tanium-specific details (server URL, auth token, etc.) and is considered an advanced enhancement rather than core to this plan.

In summary, this packaging step produces deployable artifacts for each model. By separating it from the repository sync step, you can choose to rebuild packages only when something has changed (e.g., new drivers or new HPIA version) rather than every time the sync runs. We also include support for a **"force rebuild"** flag (overriding config or via CLI) to rebuild all packages regardless of changes, useful for consistency or if something in the installer script changed and you need to repack all zips with the new script.

Logging Framework

Robust logging is vital for troubleshooting and auditing the automation process. We will implement a custom logging framework in PowerShell to cover both console output with colors and timestamps, and persistent log files with rotation. Key features of the logging system:

- **Log Levels**: We define log levels such as DEBUG (very detailed internal info), VERBOSE (more detail on actions taken), INFO (general progress), WARNING (non-critical issues), ERROR (critical problems), and SUCCESS (for highlighting completions). Each log message will be tagged with its level.
- **Console Output**: Use color-coded output to distinguish levels. For example:
- DEBUG – dark gray (or another subtle color)
- VERBOSE – gray
- INFO – white (default console color)
- SUCCESS – green
- WARNING – yellow
- ERROR – red
  PowerShell's `Write-Host` with `-ForegroundColor` can be used for coloring. We will likely create a function `Write-Log` that takes a message and a level, and internally uses `Write-Host` with the appropriate color for console. (If running non-interactively, color codes are harmless, but they ensure readability when watching a console or transcript.)
- **Timestamps**: Each log entry should be prefixed with a timestamp (e.g., `[2025-06-02 21:05:00] [INFO] Starting repository sync...`). This helps correlate events over time. We will use a consistent format, probably up to seconds. Optionally include the script name or function name if multiple scripts are writing to one file.
- **File Logging**: All log messages should also be written to a log file. This can be done by having `Write-Log` not only output to console but also append to a file (e.g., using `Add-Content` or a [System.IO.StreamWriter] for performance). The log file path can be specified in the config (e.g., `LogFilePath`). If not specified, a default like `.\logs\HP_Update.log` can be used.
- **Log Rotation**: Implement rotation to prevent the log file from growing indefinitely:
- **By size**: e.g., if the log exceeds 5 MB, roll it. Renaming the current log to something like `HP_Update.log.1` (and older ones incrementing their suffix) and starting a new file. We might keep a certain number of old logs (configurable, say 5 files).
- **By age**: e.g., rotate monthly or after X days. Alternatively, incorporate the date in the log filename (like `HP_Update_2025-06-02.log`), which implicitly rotates daily. A simple strategy: at script start, if using daily logs, open a new file with today's date.
- The config file can include settings like `MaxLogSizeMB` and `MaxLogAgeDays` or `MaxLogFiles` to control this.
- Rotation logic: check at the end of execution or start of execution if conditions are met to rotate. For size-based, checking after each write could be expensive; better to check at certain intervals or after major sections.
- **Verbose/Debug control**: The framework should respect verbosity settings. For example, by default INFO, WARNING, ERROR, SUCCESS show on console. If `-Verbose` is enabled or a config flag is set, then VERBOSE (and possibly DEBUG) messages should also appear. We can tie into PowerShell's built-in `$VerbosePreference` if desired, or manage it via our own flag.
- For instance, if `VerboseLogging` is true (from config or command-line), we output all levels >= VERBOSE. If also a `Debug` flag is given, include DEBUG level too.

- Regardless of console verbosity, **all levels should go to the log file**, so that even detailed info is preserved for troubleshooting.
- **Consistent format**: Use a consistent function or small module for logging. This avoids repetitive code and ensures all messages follow the format. We can also include a function to log exceptions (catching `$Error[0]` details) in a standardized way.
- **Color in files**: Avoid writing color codes to the file (just write plain text with level labels). Only console gets colored output.
- Possibly integrate with a tool like CMTrace format (if you have ConfigMgr background). In Ryan's blog, they used a CMTrace-compatible format (which prepends severity codes) [26], but that's optional. We can stick to a simpler approach unless a specific log viewer is preferred.

Example:

```
Write-Log -Level INFO -Message "Initialized repository for $model."
Write-Log -Level WARNING -Message "SoftPaq $spId not found on server, skipping."
Write-Log -Level ERROR -Message "Failed to install CMSL. Exiting."
```

This function would handle coloring and writing to file. Internally it might do something like:

```
function Write-Log {
    param([ValidateSet("DEBUG","VERBOSE","INFO","SUCCESS","WARNING","ERROR")]
$Level, [string] $Message)
    $timestamp = Get-Date -Format "yyyy-MM-dd HH:mm:ss"
    $logLine = "[${timestamp}] [$Level] $Message"
    # Console output with color
    switch ($Level) {
      "DEBUG"   { if ($Global:EnableDebug) { Write-Host $logLine -
ForegroundColor DarkGray } }
      "VERBOSE" { if ($Global:EnableVerbose) { Write-Host $logLine -
ForegroundColor Gray } }
      "INFO"    { Write-Host $logLine -ForegroundColor White }
      "SUCCESS" { Write-Host $logLine -ForegroundColor Green }
      "WARNING" { Write-Host $logLine -ForegroundColor Yellow }
      "ERROR"   { Write-Host $logLine -ForegroundColor Red }
    }
    # Always append to log file
    Add-Content -Path $Global:LogFile -Value $logLine
}
```

(The above is pseudo-code for illustration; actual implementation would refine how global flags are used.)

- **Log file location**: Could default to a "logs" subdirectory. Ensure the script creates the directory if not present. The location can be overridden via config (some might want it on a share or specific path).
- **Rotation implementation**: possibly a separate function `Rotate-Log` that the main script calls at startup. It would check the existing log file, and if size > threshold or if file is older than certain days (or the day changed), perform the rotation.

By implementing this logging, whenever the automation runs (whether manually or via scheduled task), there will be a detailed record. In case of errors, one can inspect the log file to see where it failed (e.g., network failure on a specific SoftPaq, or packaging error, etc.). The colored console output is helpful during interactive runs to quickly spot warnings or errors.

Additionally, this logging framework can be reused across all scripts (the CMSL installer script, the repo sync script, the package builder script, and even the client install script could use a similar smaller logging approach for local logs). Consistency in logging makes reading across different components easier.

Execution Modes

The automation should support various **execution modes** via command-line switches or parameters, allowing flexibility in how it's run. This includes dry-run simulation, verbose output, and forcing rebuilds of packages. We outline the key modes and how they influence script behavior:

- **Standard Mode**: Running the main script (or a master orchestrator that calls sub-scripts) with default settings will perform a full operation: ensure CMSL is installed, update all repositories per config, and build any needed packages. It would normally only do actual work when required (downloads only new updates, builds packages only if changed). Logs at info level by default.
- **Dry Run Mode** (`-DryRun` switch): In this mode, the script simulates the actions without making permanent changes. Use cases include testing the config and seeing what would happen (which models would sync, which updates would download) without actually downloading or writing files. Implementation:
- The script will load config and perform checks, but when it comes to invoking `Invoke-RepositorySync`, it may instead call it with `-WhatIf` (if supported) or skip it entirely and just log what it *would* do (e.g., "Would synchronize repository for model XYZ").
- Similarly for package building, instead of actually zipping, it can scan the directories and log what it *would* include and what package name *would* be created.
- Dry run should never install or remove anything. It's essentially a preview. This is helpful for change control or to ensure everything is set correctly (especially filters) before doing a potentially large download.
- Implementation detail: Use a global flag `$DryRun` set by the switch. Wrap actions with conditions, e.g.:

```
if ($DryRun) { Write-Log -Level INFO "DryRun: Skipping actual download for $repoDir" }
else { Invoke-RepositorySync ... }
```

- The output and logs should clearly mark these operations as "DryRun" so it's obvious no changes were made.
- **Verbose/Debug Mode** (`-Verbose`, `-Debug`): These leverage PowerShell's common parameters or custom switches to increase output detail.
- `-Verbose`: When user runs the script with `-Verbose`, it sets `$VerbosePreference="Continue"` which causes any `Write-Verbose` calls to output. Our logging could tie into this, or simply we check for a parameter and set our `$EnableVerbose` (as

seen in logging framework). In verbose mode, the script might log additional details at each step (like the exact command it's running, or detailed filter info).
- `-Debug`: Similar for debug, it could show extremely detailed info (like internal variable states, or detailed output from CMSL commands). This might be too granular for routine use but helpful in diagnosing issues with the script logic.
- Ensure that using these doesn't interfere with normal operation aside from extra output. (PowerShell's own -Verbose can be used in Write-Log or we handle manually as above.)
- **Force Rebuild Mode** (`-ForceRebuild` switch): This specifically forces the package rebuild step to run for all platforms, regardless of whether the repository had changes.
- Normally, we might optimize package building: e.g., if after sync, we detect that no SoftPaqs were updated for a model and the HPIA version is unchanged, we could decide to skip re-zipping that model's package (since it would be identical to last time). We could track a hash or timestamp of last change per repo. However, if the user supplies `-ForceRebuild`, we bypass that check and rebuild all zips anew.
- Use cases: ensuring that all packages are recreated (maybe to update the included install script if it changed, or to reset any issues with previous zips).
- Implementation: A simple boolean that overrides the conditional logic in the packaging function. If true, treat every repository as "modified".
- **Selective Mode** (possible enhancement): Perhaps allow running for a subset of models. For example, a parameter `-PlatformID 87EF` could limit the operation to one model's repository and package. This can be useful if you only want to update one model at a time (maybe testing a new model addition). We can implement this by filtering the loaded config's platform list to the one specified. If not provided, default is all.
- **Confirm Prompts**: By default, destructive actions (like removing old SoftPaqs or overwriting existing packages) could be preceded by a confirmation prompt. However, since this is meant to be automated (possibly run scheduled), it should run unattended. Therefore, we will use `-Force` or ensure `-Confirm:$false` on cmdlets like `Remove-Item` for cleanup, after carefully ensuring we target the right files. The dry-run mode covers the "safe preview" need, so in actual run we won't prompt.
- **Error Handling Mode**: Possibly a `-ContinueOnError` flag if we want the script to attempt to continue updating other models even if one model fails. By default, an error in one repository might not halt the whole script; we can catch exceptions around each model's sync so that it logs the error and moves to the next model, then perhaps returns a summary at end that some failed. This isn't exactly a mode but an approach — however, we could allow a strict mode where any error stops everything (for debugging). This could be controlled by a switch or by instructing users to use `-ErrorAction Stop` and try-catch.

Usage Examples: - Run everything normally:
`.\Update-HPRepositories.ps1`
(Assuming that script orchestrates CMSL install, sync, and package build according to config.) - Test what would happen without making changes:
`.\Update-HPRepositories.ps1 -DryRun -Verbose`
This would output all planned actions in detail. - Force recreation of zips after syncing:
`.\Update-HPRepositories.ps1 -ForceRebuildPackages`
(Combined with -Verbose if needed for more info). - Only update one model (if implemented):
`.\Update-HPRepositories.ps1 -PlatformID 87EF`

Each script (if we split tasks) can also have its own switches. For instance, if we have `Sync-HPRepos.ps1` and `Build-HPPackages.ps1` separately, they could share a config file and have their own `-DryRun` or `-Verbose`. A combined script just routes the parameters appropriately.

The execution modes ensure that the automation is flexible: one can simulate runs for safety, get more info when troubleshooting, or force actions when needed – all through simple switches, rather than editing the script. All these modes should be documented in the script help and README.

Documentation and Script Help

All components of this solution will include thorough documentation to assist administrators in understanding and extending the system. This includes inline script documentation (help comments) and a top-level README for the project.

Inline Script Documentation:
Each PowerShell script or module will have a comment-based help section at the top following the standard PowerShell help syntax: - **.SYNOPSIS** – A brief description of what the script or function does. (e.g., "Synchronizes HP driver/firmware repositories for configured platforms and builds deployment packages.") - **.DESCRIPTION** – A more detailed explanation. This can span multiple lines, explaining the script's purpose, context (e.g., mention HP CMSL and HPIA usage), and any important details about how it works or its requirements. - **.PARAMETER** – For each parameter or switch, describe what it does. For example, `-DryRun` would be documented as "If present, performs a simulation of actions without downloading or modifying files. Use this to see intended changes." Another example, `-PlatformID` (if present) could be "Limits the operation to the specified HP platform ID (4-digit hex). Only that platform's repo will be synced and packaged." - **.EXAMPLE** – Provide usage examples. At least a couple for typical use (as shown above in Execution Modes), and perhaps one for each important switch combination. PowerShell's help system will show these to the user if they run `Get-Help`. - **.NOTES** – (Optional) could include author name, version of the script, date, and any references. Here we might also note prerequisites like "Requires HP CMSL 1.8.2 or later, and PowerShell 5.1 or higher (PowerShell 7 recommended)."

By following this format, an admin can run `Get-Help .\Update-HPRepositories.ps1 -Detailed` and see all this information. It's crucial to keep this up-to-date as the script evolves.

README.md (Root Documentation):
A comprehensive README in Markdown will be provided alongside the scripts. This document (similar in structure to this plan, but phrased for end-users) will cover: - **Overview**: What the solution is and why it's useful (ensuring HP clients have updated drivers/firmware via Tanium using HP's official tools). - **Components**: List the scripts included (e.g., `Install-CMSL.ps1`, `Sync-HPRepos.ps1`, `Build-TaniumPackages.ps1`, `Install-HPUpdates.ps1`) and describe each module's role. - **Configuration**: Explain the JSON configuration file. Provide a section detailing each field in the config, what values are expected, and how to add a new platform. Also mention the comment support and any limitations (like requiring PS7 for JSON comments). - **Dependencies**: List requirements like PowerShell version, HP CMSL, HPIA. For example: "PowerShell 5.1 (Windows 10 1809 or later) or PowerShell 7+. HP Client Management Script Library 1.8.2 (the script will install this if not present). Internet access to HP's download sites (or a proxy configured) to download SoftPaqs. Approximately X GB of disk space per platform repository (varies by model). Tanium Module (optional: if using Tanium API integration)." - **Usage Guide**: Step-by-step

instructions on how to run the scripts: 1. Configure the JSON file with your platforms and preferences. 2. Run the CMSL installer script (or the main script which will install CMSL automatically). 3. Run the sync script to build/update the repositories. 4. Run the package builder (or main script with appropriate flags) to create the zip packages. 5. Import those zips into Tanium or copy them to the Tanium packages directory, and create a deployment package in Tanium that executes the provided install script. 6. Optionally, schedule the sync script to run periodically (e.g., via Task Scheduler or a CI pipeline) so that new drivers are fetched regularly. After each sync, run the package build to update the deployment packages in Tanium. - **Tanium Deployment**: How to use the produced packages in Tanium. For example: "In Tanium Console, create a new package. Upload the zip file (or host it on a share if Tanium uses that method), and add the install script as the command to execute (e.g., `powershell -ExecutionPolicy Bypass -File Install-HPUpdates.ps1`). Ensure the package is targeted appropriately (e.g., with a targeting filter for the specific model if you separated packages per model, or a filter within the script)." This part may also note that one can use Tanium's library or API if automating package import. - **Extending the Solution**: Notes on how to add new categories (if say you later want to include HP software or UWP apps), how to adjust filters, or include new script functionality. Perhaps also how to adapt it if needed to similar tools for other vendors (though out of scope, it shows forward-thinking). - **Troubleshooting**: Common issues and resolutions. For example: - "CMSL fails to install" – check you are running as admin, .NET requirement, etc. - "Repository sync slow or times out" – perhaps increase timeout or run off hours. - "No updates found for a model" – maybe the platform ID is wrong or HP hasn't released any for that OS. - "Tanium deployment didn't install anything" – ensure the install script ran as expected, check HPIA logs, etc. - Provide guidance that logs are available (give path from config) and to inspect them. - **Security Considerations**: reiterate some points from recommendations like verifying downloads, using code signing if possible, and principle of least privilege (though these scripts require admin to run, especially for installing drivers). - **References**: Optionally, link to HP documentation such as the CMSL user guide or HPIA user guide, for further reading [27] [23]. Also, if any third-party tools like 7-Zip are used, mention that.

Overall, the documentation ensures that someone new to this solution can set it up and maintain it. The README serves as the primary guide, while the script help provides quick in-context assistance. All documentation should be maintained when changes are made to the scripts.

By providing both inline help and a separate documentation file, we cover both the scenario where an admin is reading the project on a repository (seeing README) and when they are directly using the script (using Get-Help). This reduces confusion and support effort.

Recommendations and Best Practices

Finally, we outline additional recommendations to improve the solution's reliability, security, and maintainability:

- **Security Controls**:
- **Verify Downloads**: Always verify the integrity of external downloads like the CMSL installer and HPIA. Use cryptographic hashes (if HP publishes SHA256 hashes for these tools on their site, compare against those) or at least verify the digital signature publisher (`Get-AuthenticodeSignature` in PowerShell can ensure the signer is HP Inc.) as a safety check before executing these tools.

- **HTTPS and Proxy**: Ensure that the system uses HTTPS (which hpcloud URLs do) for downloads to prevent tampering. If your environment uses a web proxy, make sure PowerShell's web requests go through it (so they can be monitored/controlled).
- **Principle of Least Privilege**: Run the packaging scripts on a secure system with only the necessary access. The script itself will need admin rights (to install CMSL and possibly to run HPIA if doing BIOS updates), but it should be executed by trusted users or as a scheduled task under a service account. Limit the access to the output files (zips) since they contain driver executables – store them in a controlled location to prevent unauthorized modification before they get to Tanium.
- **Code Signing**: Consider code-signing the PowerShell scripts themselves. This ensures that in environments with execution policy set to AllSigned, the scripts can run. It also provides assurance that the scripts haven't been altered maliciously. If code signing is not possible, at least use `RemoteSigned` policy (as the blog suggests enabling scripts with RemoteSigned [28]).
- **Execution Policy and Bypass**: Our install script for endpoints uses `-ExecutionPolicy Bypass` when run via Tanium to ensure it runs, but ensure your organization is aware and approves that (since it's a controlled script we provide).
- **Audit Logging**: The logs we generate can be considered sensitive (they list actions and possibly URLs or file paths). Protect the log files, and consider forwarding them to a logging system or SIEM for centralized monitoring of update actions. Unusual events (like an unexpected error, or an update that includes firmware) might warrant attention.
- **Cleanup**: Regularly cleanup old artifacts. The repository sync already removes superseded SoftPaqs. Also consider cleaning out extremely old models if they are decommissioned (remove their config entry and archive/delete their repo folder to save space). The packaging script could also remove old package files if not needed. For example, if you keep the last 2 versions of each zip just in case, you could clean older ones.
- **Best Practices for CMSL/HPIA**:
- Always use the **latest CMSL and HPIA** versions, as they are updated to support new models and OS releases. Our automation makes this easy by checking versions. Keep an eye on HP's updates (perhaps subscribe to notifications or regularly check HP's site) to update the config for new CMSL/HPIA releases.
- Use **appropriate filters** to avoid bloat: e.g., if you don't want optional software in your deployment, exclude the "Software" category [7]. If you only want critical updates in a production environment, maybe exclude "Routine". This keeps packages smaller and focused. You can maintain a separate config or run for different categories if needed (like maybe once a quarter include Routine).
- **Test BIOS updates carefully**: If including BIOS in the updates, be mindful that they may require special handling (BIOS updates often require a restart and possibly a BIOS password if set on the device). HPIA and CMSL can handle BIOS updates, but if your PCs have BIOS Setup passwords, you'll need to provide that for automated updates. HPIA supports a `/BIOSPwdFile` parameter (with an encrypted password file) – you might incorporate that by distributing a password file or by having the endpoint script call CMSL's `Set-HPBIOSPassword` if available. This is an advanced scenario; at minimum, mention in docs that BIOS updates are included and require either no password or additional steps.
- **Reboot Management**: Consider how to handle reboots after updates. Some driver installs or especially BIOS/firmware will need reboots. Decide if Tanium should flag the device for reboot or if the script should trigger it. It might be safer to let Tanium or administrators handle scheduling reboots (especially if doing many machines at once, you don't want them all rebooting immediately). Possibly log a message "Reboot required: Yes" if HPIA indicates so, and let Tanium pick that up.
- **Enhancements**:

- **Parallel Downloads**: If you have many models and a fast internet connection, the sync script could download for multiple models in parallel to speed up the overall runtime. HPCMSL might not natively support parallel within one process, but you could start separate background jobs for different models. Ensure the system has resources and avoid hitting HP's servers too hard. Usually, updating sequentially overnight is fine, but this is an option.
- **Progress Reporting**: Implement some progress output for long operations. For example, after each model's repo sync, print a summary (e.g., "Model X: 2 new SoftPaqs downloaded, 10 total SoftPaqs in repository"). During zipping, maybe output the percent done if possible. This is mainly for interactive use; if running unattended, rely on logs.
- **Tanium API Integration**: As mentioned, you could extend the packaging script to automatically update Tanium with the new package. Tanium has REST APIs that allow uploading files and creating packages. This would fully automate the pipeline: run the script, and your Tanium deployment packages are updated without manual steps. If implementing, ensure you handle authentication securely (probably using an API user and storing credentials safely).
- **Cross-Vendor Support**: While out of scope for now, this framework could be extended for other OEMs (Dell, Lenovo) which have their own update catalogs and tools. Designing with modularity (maybe abstracting "HP CMSL" as a provider) could allow future inclusion of those. For now, focusing on HP, ensure the design doesn't hard-code too many assumptions that prevent adding such functionality later.
- **Environment Validation**:
- At script start, perform a check of the environment: OS version (ensure the host OS is supported by CMSL; HPCMSL supports Windows 10 1809+ and Windows 11 [29]), PowerShell version (5.1 or higher), and required modules.
- If something is not right (e.g., script run on Windows 8 or PS5.0), clearly log and abort with an explanation.
- Also, ensure there is sufficient disk space on the drive holding `RepositoryBase` before syncing (maybe log a warning if free space is below a threshold because driver repositories can be tens of GB).
- If using 7-Zip, ensure it's installed or the path is configured. If not, log a notice that default compression will be used.
- **Testing**:
- Test the process in a lab environment for one model first. Verify that the repository is correctly populated (check the `.repository\activity.log` CMSL produces for any errors [30]). Test that the Tanium package installs drivers correctly on a test machine (look at HPIA's report to ensure all applicable updates were installed).
- Once validated, roll out to more models. Use the script's ability to run per model for targeted testing if available.
- Keep an eye on log files for any recurring warnings (for example, if a certain SoftPaq consistently fails to download, you might investigate if HP removed it or if credentials are needed).

Implementing these best practices and recommendations will help maintain a robust and secure update deployment process. This solution leverages official HP tools (CMSL and HPIA) which ensures that only vendor-approved updates are applied (no third-party driver sources), and by automating it with PowerShell, it reduces manual effort and potential for error. Continual improvement and vigilance (like monitoring for new HP releases, and adjusting config or scripts accordingly) will ensure the solution remains effective over time.

---

1 Deploying HP Client Management Script Library (HPCMSL) with Microsoft Endpoint Manager | hp's Developer Portal
https://developers.hp.com/hp-client-management/blog/deploying-hp-client-management-script-library-hpcmsl-microsoft-endpoint-manager

2 Silent Installation of HP Client Management Script Library (1.7.1)
https://www.manageengine.com/products/desktop-central/software-installation/silent_install_HP-Client-Management-Script-Library-(1.7.1).html

3 Initialize-Repository | hp's Developer Portal
https://developers.hp.com/hp-client-management/doc/initialize-repository

4 5 6 7 8 9 Add-RepositoryFilter | hp's Developer Portal
https://developers.hp.com/hp-client-management/doc/add-repositoryfilter

10 11 12 13 Set-RepositoryConfiguration | hp's Developer Portal
https://developers.hp.com/hp-client-management/doc/set-repositoryconfiguration

14 15 16 17 26 30 Automating HP SoftPaq Repository Updates Using PowerShell – Ryan's Tech Blog
https://ryandengstrom.com/2019/04/09/automating-hp-softpaq-repository-updates-using-powershell/

18 19 20 HP Image Assistant | HP Client Management Solutions
https://ftp.ext.hp.com/pub/caps-softpaq/cmit/HPIA.html

21 27 28 Automating Maintenance of HP Image Assistant Repositories (Upd Feb 27, 2024) | hp's Developer Portal
https://developers.hp.com/hp-client-management/blog/automating-maintenance-hp-image-assistant-repositories-upd-feb-27-2024

22 ConvertFrom-Json (Microsoft.PowerShell.Utility) - PowerShell | Microsoft Learn
https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.utility/convertfrom-json?view=powershell-7.5

23 24 HP Image Assistant User Guide
https://ftp.hp.com/pub/caps-softpaq/cmit/whitepapers/HPIAUserGuide.pdf

25 HP Image Assistant new /Force switch | hp's Developer Portal
https://developers.hp.com/hp-client-management/blog/hp-image-assistant-new-force-switch

29 Client Management Script Library | hp's Developer Portal
https://developers.hp.com/hp-client-management/doc/client-management-script-library