Ultimate PowerShell Command Guide

A comprehensive day-to-day reference for PowerShell scripting syntax, core cmdlets, legacy Windows CLI tools, and the major sysadmin modules (Active Directory, Microsoft Graph, Azure, and Exchange Online). Companion guide to the Ultimate Linux Command Guide.

Windows CLI Commands

These are legacy Windows command-line utilities that predate PowerShell (many going back to MS-DOS or Windows NT) but are still run daily by sysadmins, either directly in cmd.exe, from a PowerShell prompt (PowerShell can call native .exe tools directly), or embedded inside batch/PowerShell scripts. Several of these still don't have a fully equivalent PowerShell cmdlet, which is exactly why they've survived this long.

- **ipconfig** = displays the IP configuration for all network adapters on the local machine — the Windows equivalent of Linux's `ip addr`.
  - **/all** = shows full configuration detail (DNS servers, DHCP lease info, MAC address, WINS) instead of the abbreviated default view.
  - **/release** and **/renew** = releases or renews the current DHCP-assigned IP address lease.
  - **/flushdns** = clears the local DNS resolver cache — the single most common first troubleshooting step for stale DNS records on a client machine.
  - **/displaydns** = shows the current contents of the local DNS resolver cache.
  - **/registerdns** = manually refreshes DHCP leases and re-registers DNS names, useful for a client that has fallen out of sync with Dynamic DNS.
- **ping** = tests basic network reachability using ICMP Echo packets, identical in purpose to the Linux `ping` command.
  - **-t** = pings continuously until manually stopped with Ctrl+C (Linux's `ping` does this by default; Windows requires this flag since its default is only 4 pings).
  - **-n \<count\>** = specifies the number of echo requests to send.
  - **-l \<size\>** = sets the packet payload size in bytes, useful for testing for MTU/fragmentation issues.
  - **-4 / -6** = forces IPv4 or IPv6 specifically.
  - **-a** = resolves addresses to hostnames in the output.
- **tracert** = maps the network path (each router hop) between the local machine and a destination host, using ICMP by default — the Windows equivalent of Linux's `traceroute`.
  - **-d** = does not resolve IP addresses to hostnames, which speeds up the trace considerably.
  - **-h \<max hops\>** = sets the maximum number of hops to trace before giving up.
  - **-w \<timeout\>** = sets the wait time in milliseconds for each reply before timing out that hop.
- **pathping** = combines the hop-mapping of `tracert` with the ongoing statistical sampling of `ping`, running an extended test against every hop along the route to identify exactly where packet loss or latency is occurring — there isn't a single direct Linux equivalent, though it's philosophically similar to `mtr`.
  - **-n** = does not resolve hostnames, speeding up the test.
  - **-q \<count\>** = sets the number of queries sent per hop (default is 100, which is why pathping takes noticeably longer than tracert).
- **nslookup** = queries DNS servers for hostname/IP resolution and specific record types, exactly like its Linux namesake.
  - Non-interactive: `nslookup hostname.domain.com` for a quick single lookup.

    - Interactive mode: running bare `nslookup` drops you into a shell where you can run `set type=MX`, `set type=NS`, `server \<ip\>` to change which DNS server you're querying, etc.
    - **-type=\<record\>** = specifies the DNS record type to query (A, AAAA, MX, NS, SOA, TXT, CNAME, ANY) as a command-line flag before entering interactive mode.
- **netstat** = displays active network connections and listening ports on the local machine — the direct Windows equivalent of Linux's `ss`/`netstat`.
    - **-a** = shows all connections and listening ports.
    - **-n** = shows addresses and port numbers numerically, skipping hostname resolution (much faster).
    - **-o** = includes the owning Process ID (PID) for each connection, letting you cross-reference with Task Manager or `tasklist` to find which application owns a port.
    - **-b** = shows the executable name involved in creating each connection (requires elevated/admin privileges).
    - **-p \<protocol\>** = filters output to a specific protocol (TCP, UDP, TCPv6, UDPv6).
    - A very common combo: `netstat -ano | findstr :443` to find exactly what's listening on port 443.
- **netsh** = network shell — an extensive, context-based configuration tool for networking components (interfaces, firewall, WLAN, routing). It's organized into 'contexts' you navigate into, similar to how nmcli uses 'objects' on Linux.
    - **netsh advfirewall firewall show rule name=all** = lists every Windows Defender Firewall rule currently configured.
    - **netsh advfirewall firewall add rule name="Allow RDP" dir=in action=allow protocol=TCP localport=3389** = adds a new inbound firewall rule from the command line.
    - **netsh interface ip set address** = sets a static IP address on a network interface from the command line (largely superseded by the `New-NetIPAddress` PowerShell cmdlet, but still very common in legacy scripts).
    - **netsh wlan show profiles** = lists saved Wi-Fi network profiles on the machine.
    - **netsh int ip reset** and **netsh winsock reset** = resets the entire TCP/IP stack or Winsock catalog to defaults — a classic 'nuke it and reboot' fix for deeply broken networking.
- **systeminfo** = dumps a detailed configuration summary of the local machine: OS version/build, installed hotfixes, BIOS version, total/available physical memory, domain, and boot time — a quick one-shot equivalent of combining several Linux commands (`uname -a`, `dmidecode`, `free`).
    - **/s \<computer\>** = queries a remote computer instead of the local one (requires appropriate remote access rights).
    - This command can be noticeably slow because it actively checks for hotfix and update information; for a faster subset, `Get-ComputerInfo` in PowerShell is often preferred for scripting.
- **tasklist** = lists all currently running processes on the local (or a remote) machine, along with PID, memory usage, and session information — the classic cmd.exe equivalent of Linux's `ps` or PowerShell's `Get-Process`.
    - **/v** = verbose output, including the window title and CPU time for each process.
    - **/svc** = lists the Windows services hosted inside each `svchost.exe` process — extremely useful for identifying which service a specific svchost instance is actually running.
    - **/fi "\<filter\>"** = filters results (ex. `tasklist /fi "imagename eq notepad.exe"`).
    - **/s \<computer\>** = queries a remote computer.
- **taskkill** = forcibly terminates a running process by PID or image name — the cmd.exe equivalent of Linux's `kill`/`pkill`.
    - **/pid \<id\>** = targets a specific Process ID.
    - **/im \<name\>** = targets by image/executable name (ex. `taskkill /im notepad.exe`), and can match multiple running instances at once.

    - **/f** = force terminate (the equivalent of `kill -9`/SIGKILL, versus a normal request which is closer to a graceful SIGTERM).
    - **/t** = tree-kill; also terminates all child processes spawned by the target process.
- **sc** = service controller — the low-level cmd.exe utility for querying and configuring Windows services, predating `Get-Service`/`Set-Service` in PowerShell but still commonly used for options those cmdlets don't expose directly.
    - **sc query \<service\>** = shows the current status of a service.
    - **sc qc \<service\>** = query configuration; shows the service's startup type, binary path, and dependencies.
    - **sc config \<service\> start= auto|demand|disabled** = changes a service's startup type (note the required space after the equals sign — a classic gotcha).
    - **sc start / sc stop \<service\>** = starts or stops a service.
    - **sc failure \<service\> reset= 86400 actions= restart/60000** = configures automatic recovery actions if a service crashes (a capability `Set-Service` in older PowerShell versions couldn't configure at all).
- **reg** = the command-line interface for querying and modifying the Windows Registry, functioning as a scriptable alternative to the regedit.exe GUI.
    - **reg query \<keypath\>** = reads a registry key or value (ex. `reg query "HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion"`).
    - **reg add \<keypath\> /v \<name\> /t \<type\> /d \<data\>** = adds or updates a registry value. `/t` specifies the data type (REG_SZ, REG_DWORD, REG_BINARY, etc.).
    - **reg delete \<keypath\> /v \<name\>** = deletes a specific registry value (or the whole key if `/v` is omitted, with `/f` to force it without confirmation).
    - **reg export / reg import** = backs up a registry key to a `.reg` file, or restores one from a file — a common pre-change safety step before editing production registry settings.
    - **/s** = a common flag when querying, recursively searches subkeys as well.
- **robocopy** = Robust File Copy — a powerful, resilient file/directory copy utility built for reliably mirroring large datasets, resuming after network interruptions, and preserving file attributes/permissions/timestamps. Far more capable than the legacy `xcopy` and the closest Windows analog to `rsync`.
    - **/e** = copies all subdirectories, including empty ones.
    - **/mir** = mirrors a directory tree exactly — copies everything from source to destination **and deletes** files at the destination that no longer exist at the source (the direct equivalent of `rsync --delete`; use with caution).
    - **/z** = restartable mode; if the copy is interrupted mid-file over an unstable network connection, it can resume from where it left off instead of starting over.
    - **/copyall** = copies all file information: data, attributes, timestamps, owner, and NTFS permissions/auditing info.
    - **/r:\<n\>** = sets the number of retries on a failed copy (the default of 1,000,000 retries is notorious for making a script appear to hang on a locked file — always set this explicitly in scripts).
    - **/w:\<seconds\>** = sets the wait time between retries (also worth explicitly lowering from its long default in scripts).
    - **/log:\<file\>** = writes output to a log file instead of (or in addition to) the console; `/log+:` appends instead of overwriting.
- **xcopy** = the legacy, simpler predecessor to robocopy for copying files and directory trees. Still occasionally used for quick, simple copy jobs in older scripts, though robocopy is the modern recommended tool for anything nontrivial.
    - **/s** = copies directories and subdirectories, except empty ones (`/e` also copies empty ones).
    - **/h** = copies hidden and system files as well.
    - **/y** = suppresses the overwrite confirmation prompt.

- **wmic** = Windows Management Instrumentation Command-line — a legacy command-line front end for querying WMI data (hardware, OS, installed software, processes) directly from cmd.exe. **Deprecated by Microsoft** in favor of PowerShell's `Get-CimInstance`, but still frequently found in older scripts and worth recognizing.
    - **wmic os get caption,version,buildnumber** = queries basic OS version info.
    - **wmic product get name,version** = lists installed MSI-based software (notoriously slow, since it silently triggers a consistency check of the whole MSI installer database).
    - **wmic process where name="notepad.exe" call terminate** = kills a process by name.
    - **wmic logicaldisk get size,freespace,caption** = shows disk space per drive letter.
    - The modern replacement is `Get-CimInstance -ClassName Win32_OperatingSystem` (or the equivalent `Win32_Product`, `Win32_Process`, `Win32_LogicalDisk` classes), covered in the Core Cmdlets section.
- **gpupdate** = forces the local machine to immediately re-check and re-apply Group Policy settings from the domain, instead of waiting for the normal background refresh interval.
    - **/force** = reapplies **all** policy settings, not just the ones that have changed since the last refresh — the standard troubleshooting step after making a GPO change you need to test immediately.
    - **/target:computer** or **/target:user** = limits the refresh to only computer-scoped or only user-scoped policy settings.
- **gpresult** = displays the Resultant Set of Policy (RSoP) for a user or computer — showing exactly which Group Policy Objects actually applied (and which were filtered out, and why), which is essential for troubleshooting 'why isn't this policy taking effect' tickets.
    - **/r** = shows a summary report of applied GPOs.
    - **/h \<file.html\>** = generates a detailed HTML report, which is dramatically easier to read than the console output for complex policy troubleshooting.
    - **/scope:computer** or **/scope:user** = limits the report to only computer or only user policy results.
- **whoami** = displays the identity of the currently logged-in user — identical in purpose to the Linux command of the same name.
    - **/groups** = lists every security group the current user is a member of, including nested/domain groups.
    - **/priv** = lists the current user's assigned privileges (e.g., SeDebugPrivilege, SeBackupPrivilege) and whether each is currently enabled.
    - **/all** = combines user, group, and privilege information into one comprehensive dump.
- **net user / net group / net localgroup** = the legacy command-line family for managing local and domain user accounts and groups from cmd.exe, predating the ActiveDirectory PowerShell module.
    - **net user \<username\> \<password\> /add** = creates a new local user account.
    - **net user \<username\> /domain** = queries a domain account's basic info (lockout status, last logon, password expiration).
    - **net localgroup Administrators \<username\> /add** = adds a user to a local group (most commonly the local Administrators group).
    - **net group "\<GroupName\>" \<username\> /add /domain** = adds a user to a domain security group.
    - **net accounts** = displays or configures domain-wide password policy settings (min length, lockout threshold, max age).
- **shutdown** = restarts, shuts down, or logs off the local (or a remote) machine from the command line — used heavily in scheduled maintenance scripts.
    - **/r** = restart. **/s** = full shutdown. **/l** = log off.
    - **/t \<seconds\>** = sets a delay before the action executes (`/t 0` executes immediately).
    - **/m \\computername** = targets a remote computer instead of the local one.
    - **/a** = aborts a pending scheduled shutdown/restart.

    - **/f** = forces running applications to close without warning users first.
- **sfc** = System File Checker — scans and repairs corrupted or missing Windows system files by comparing them against a protected cached copy. Roughly analogous in *purpose* to running an integrity check on core OS packages in Linux, though the mechanism is entirely different.
    - **/scannow** = performs an immediate full scan and automatically repairs any protected files it finds corrupted. Must be run from an elevated command prompt.
- **DISM** = Deployment Image Servicing and Management — services and repairs the underlying Windows component store (WinSxS), and is frequently used **before** `sfc /scannow` if the component store itself is corrupted (since `sfc` pulls its 'known good' copies from that same store).
    - **/Online /Cleanup-Image /CheckHealth** = a quick, lightweight check for known component store corruption flags.
    - **/Online /Cleanup-Image /ScanHealth** = a deeper scan of the actual component store for corruption (slower than CheckHealth).
    - **/Online /Cleanup-Image /RestoreHealth** = attempts to repair detected corruption, typically pulling replacement files from Windows Update (or a specified local source with `/Source:`).
- **chkdsk** = Check Disk — scans a volume for filesystem errors and (optionally) bad physical sectors, and can repair filesystem inconsistencies — the closest Windows equivalent to Linux's `fsck`.
    - **/f** = fixes filesystem errors found on the disk.
    - **/r** = locates bad sectors and attempts to recover readable information from them (implies `/f`).
    - If the target volume is in use (e.g., the system drive), chkdsk cannot run immediately and instead schedules itself to run automatically on the next reboot.
- **diskpart** = a text-mode, interactive disk partitioning utility — the Windows equivalent in *purpose* to Linux's `fdisk`/`parted`, used for creating/deleting partitions, extending volumes, and assigning drive letters, all from an interactive prompt or a scripted list of commands.
    - **list disk / list volume / list partition** = lists disks, volumes, or partitions respectively so you can select the correct target.
    - **select disk \<n\>** = selects a specific disk as the active context for subsequent commands (mirrors the concept of selecting a target device in `fdisk`).
    - **create partition primary** = creates a new primary partition on the selected disk.
    - **extend size=\<MB\>** = extends the selected volume's size, e.g., after growing a virtual disk at the hypervisor level.
    - **assign letter=\<L\>** = assigns a drive letter to the selected volume.
    - **clean** = wipes the partition table of the selected disk (DANGEROUS — destroys all partition data on that disk).
- **schtasks** = the command-line interface for creating, querying, and deleting Windows Task Scheduler jobs — the cmd.exe equivalent of Linux's `crontab`, though modern scripting typically prefers the `Register-ScheduledTask` family of PowerShell cmdlets instead.
    - **/create /tn \<name\> /tr \<program\> /sc \<schedule\>** = creates a new scheduled task. `/sc` accepts schedule types like DAILY, WEEKLY, ONSTART, ONLOGON, MINUTE.
    - **/query /tn \<name\>** = shows the status/configuration of a scheduled task.
    - **/run /tn \<name\>** = manually triggers a scheduled task to run immediately, regardless of its normal schedule.
    - **/delete /tn \<name\> /f** = deletes a scheduled task without a confirmation prompt.
- **wevtutil** = queries and manages Windows Event Logs directly from the command line — a lower-level, faster alternative to the PowerShell `Get-WinEvent`/`Get-EventLog` cmdlets, often used inside lightweight monitoring scripts.

    - **el** (enum-logs) = lists all available event log channels on the system.
    - **qe \<logname\> /f:text** = queries events from a specific log and prints them as readable text.
    - **cl \<logname\>** = clears all events from a specific log (often paired with `/bu:\<path\>` to back it up to a file first).
    - **gli \<logname\>** = get-log-info; shows metadata about a log, such as its maximum size and current record count.
- **nltest** = a diagnostic tool for troubleshooting Active Directory domain trust relationships, secure channel status, and domain controller discovery — a staple tool for any AD-related connectivity issue.
    - **/dsgetdc:\<domain\>** = locates a domain controller for the specified domain (similar in purpose to how Linux's `realm discover` locates DCs via DNS SRV records).
    - **/sc_query:\<domain\>** = checks the secure channel status between the local machine and a domain controller — the classic first check when a machine reports 'trust relationship failed' errors.
    - **/sc_reset:\<domain\>** = resets a broken secure channel trust relationship without requiring a full domain rejoin.
- **dcdiag** = Domain Controller Diagnostic — runs a comprehensive suite of automated health checks against a domain controller (replication, DNS, services, connectivity, SYSVOL) and is typically the very first tool you run when a domain controller is behaving unexpectedly.
    - **/v** = verbose output with additional diagnostic detail.
    - **/test:\<testname\>** = runs only a specific named test instead of the full default suite.
    - **/s:\<DCName\>** = targets a specific remote domain controller rather than the local machine.
- **repadmin** = Replication Administration — a deep diagnostic tool specifically for inspecting and troubleshooting Active Directory replication between domain controllers, exactly the kind of tool you'd reach for during a multi-DC migration to confirm replication health.
    - **/replsummary** = gives a quick, high-level summary of replication health and any failures across all DCs.
    - **/showrepl \<DCName\>** = shows detailed inbound replication status/partners for a specific domain controller.
    - **/syncall \<DCName\>** = forces an immediate replication sync from the specified DC to its partners, instead of waiting for the normal replication interval.
- **klist** = displays currently cached Kerberos tickets on the local machine, and can also purge them — the Windows equivalent in purpose to Linux's `klist`/`kdestroy` combination for the Kerberos ticket cache.
    - **klist tickets** (or bare `klist`) = lists all Kerberos tickets currently cached for the logged-in session.
    - **klist purge** = clears all cached Kerberos tickets, forcing them to be re-requested on next use — a common fix for stale group membership/permission issues after a recent AD change.
- **certutil** = a versatile command-line certificate services utility used for dumping certificate/CRL info, managing the local certificate store, and generating file hashes.
    - **-hashfile \<file\> SHA256** = generates a hash of a file (a quick built-in alternative to PowerShell's `Get-FileHash` when you're stuck in a bare cmd.exe environment).
    - **-store \<storename\>** = lists certificates in a specified certificate store.
    - **-dump \<certfile\>** = displays detailed information about a certificate file.
    - **-urlcache -split -f \<url\>** = a lesser-known but classic trick for downloading a file from a URL directly from cmd.exe on systems without curl/Invoke-WebRequest available.
- **bcdedit** = Boot Configuration Data editor — views and modifies the boot configuration store that controls how Windows boots (boot order, timeout, safe mode flags), replacing the old boot.ini file used on legacy BIOS systems.
    - **/enum** = lists current boot configuration entries.

    - **/timeout \<seconds\>** = sets how long the boot menu waits before automatically booting the default entry.
    - **/set {current} safeboot minimal** = configures the system to boot into Safe Mode on the next restart (must be manually removed afterward with `/deletevalue {current} safeboot` or the system keeps booting to Safe Mode).
- **msiexec** = the command-line engine for installing, modifying, and removing Windows Installer (.msi) packages — heavily used in silent/unattended software deployment scripts.
    - **/i \<package.msi\>** = installs a package (normal, with UI).
    - **/x \<package.msi\>** = uninstalls a package.
    - **/quiet** or **/qn** = performs the install/uninstall completely silently with no user interface — essential for automated deployment via Intune, SCCM, or a login script.
    - **/norestart** = suppresses an automatic reboot even if the installer requests one.
    - **/l*v \<logfile\>** = generates a fully verbose install log, the first thing to check when a silent MSI install is failing mysteriously.
- **icacls** = displays and modifies NTFS file/folder permission ACLs from the command line — the Windows equivalent in purpose to Linux's `setfacl`/`getfacl` combination, but working with a fundamentally different (ACL-based rather than owner/group/other) permission model.
    - **icacls \<path\>** = displays the current ACL for a file or folder.
    - **/grant \<user\>:(rights)** = grants specific rights to a user or group (e.g., `(F)` full control, `(M)` modify, `(RX)` read & execute).
    - **/remove \<user\>** = removes all explicit permission entries for a specific user/group.
    - **/inheritance:r** = removes inherited permissions from a folder, converting them to explicit entries (or removing them entirely if combined with removing the explicit copies too).
    - **/reset** = resets permissions on a file/folder tree back to its inherited defaults.
    - **/t** = applies the operation recursively to all files and subfolders.
- **takeown** = forcibly takes ownership of a file or folder — most often used when a file's current owner (perhaps a deleted account, or SYSTEM) is preventing you from being able to change its permissions with `icacls`.
    - **/f \<path\>** = specifies the target file/folder.
    - **/r** = applies recursively to subdirectories and files.
    - **/a** = assigns ownership to the local Administrators group instead of the current user.
- **cipher** = manages NTFS file encryption (EFS) and, notably, can also securely overwrite deleted/free disk space to prevent forensic recovery of previously deleted files — conceptually similar in that second use case to Linux's `shred`, but operating at the volume's free-space level rather than on a single file.
    - **/w:\<path\>** = wipes (overwrites) all deallocated/free space on the specified drive to prevent recovery of previously deleted data.
    - **/e** and **/d** = encrypts or decrypts a specific file/folder using EFS.
- **vssadmin** = Volume Shadow Copy Service administration — manages Windows' built-in point-in-time snapshot mechanism (Volume Shadow Copies), used both for 'Previous Versions' file restore and as the underlying technology many backup tools rely on for application-consistent backups.
    - **list shadows** = lists existing shadow copies (snapshots) on the system.
    - **list shadowstorage** = shows how much disk space is currently allocated/used for shadow copy storage per volume.
    - **create shadow /for=C:** = manually creates a new shadow copy of the specified volume.
    - **resize shadowstorage /for=C: /on=C: /maxsize=20GB** = adjusts how much disk space shadow copies are allowed to consume on a volume.

- **wusa** = Windows Update Standalone Installer — installs (or uninstalls) a standalone Windows Update package (.msu file) from the command line, commonly used to deploy a specific hotfix/KB across a fleet without waiting for it through normal Windows Update/WSUS delivery.
    - **wusa \<package.msu\> /quiet /norestart** = silently installs an update package without a restart.
    - **/uninstall /kb:\<number\>** = uninstalls a specific update by its KB article number.
- **powercfg** = queries and manages Windows power plan settings from the command line, and can also generate detailed battery/sleep diagnostic reports — commonly used when standardizing power settings across desktops or laptops via a login script or GPO.
    - **/list** = lists all available power plans/schemes and shows which one is currently active.
    - **/setactive \<GUID\>** = switches the active power plan to the specified scheme.
    - **/energy** = runs a diagnostic scan and generates an HTML report identifying energy-efficiency and battery-life problems on the machine.
    - **/hibernate on|off** = enables or disables hibernation support (and the associated hiberfil.sys file consuming disk space).
- **route** = displays and modifies the local IP routing table — the Windows equivalent of the routing portion of the Linux `ip route` command.
    - **print** = displays the current routing table.
    - **add \<destination\> mask \<subnetmask\> \<gateway\>** = adds a static route.
    - **delete \<destination\>** = removes a static route.
    - **-p** = flag used with `add` to make the route persistent across reboots (otherwise it's lost on restart).
- **arp** = displays and manipulates the local ARP cache (IP-to-MAC address mappings), identical in purpose to the Linux `arp` command.
    - **-a** = displays the current ARP cache entries.
    - **-d \<IP\>** = deletes a specific entry from the ARP cache.
    - **-s \<IP\> \<MAC\>** = adds a static ARP entry.
- **telnet** = a legacy remote terminal client, largely obsolete for its original purpose but still occasionally used purely as a quick, no-frills way to test whether a specific TCP port is open/reachable on a remote host (an older habit now mostly replaced by `Test-NetConnection -Port` in PowerShell). Not installed by default on modern Windows — must be enabled first via 'Turn Windows features on or off' or `Enable-WindowsOptionalFeature`.
- **dsquery / dsget / dsadd / dsmod / dsrm** = the original legacy command-line family for querying and managing Active Directory objects, dating back to Windows Server 2003 — predating the ActiveDirectory PowerShell module by years but still occasionally seen in older logon scripts and documentation. Each tool works on LDAP distinguished names rather than the friendlier identity objects the AD module accepts.
    - **dsquery user -name "asa*"** = searches for user objects matching a pattern, returning their distinguished names.
    - **dsget user \<DN\> -memberof** = retrieves a specific property (like group memberships) for an object found via dsquery — these two are frequently piped together: `dsquery user -name "asample" | dsget user -memberof`.
    - **dsadd user \<DN\> -pwd \<password\>** = creates a new user object.
    - **dsmod user \<DN\> -disabled yes** = modifies an existing object's attributes.
    - **dsrm \<DN\>** = deletes an object.
    - In virtually every case today, the equivalent ActiveDirectory PowerShell cmdlet (`Get-ADUser`, `Set-ADUser`, etc.) is easier to read, easier to filter, and returns proper objects instead of plain text — dsquery/dsget are worth recognizing when you encounter them in a legacy script, not for writing new ones.

- **csvde** = CSV Directory Exchange — a bulk import/export tool for Active Directory objects using CSV files, useful for quick bulk exports or one-time bulk imports of new objects, though it **cannot modify or delete** existing objects (only create new ones and export existing ones).
    - **-f \<file\> -r \<LDAPfilter\>** = exports objects matching an LDAP filter to a CSV file (ex. `csvde -f users.csv -r "(objectClass=user)"`).
    - **-i -f \<file\>** = imports (creates) new objects from a properly formatted CSV file.
    - For bulk **modifications** to existing objects, `ldifde` (below) or a PowerShell loop using `Import-Csv | ForEach-Object { Set-ADUser ... }` are the correct tools instead.
- **ldifde** = LDAP Data Interchange Format Directory Exchange — similar in purpose to csvde but uses the more flexible LDIF file format and, critically, **can modify and delete** existing AD objects in bulk, not just create new ones.
    - **-f \<file\> -r \<LDAPfilter\>** = exports matching objects to an LDIF file.
    - **-i -f \<file\>** = imports an LDIF file, which can contain a mix of add, modify, and delete operations based on the `changetype` field defined for each record.
- **setspn** = manages Service Principal Names (SPNs) on Active Directory accounts — SPNs are what allow Kerberos authentication to correctly identify which service a client is trying to authenticate to (e.g., an MSSQL instance or a custom web app service account). Misconfigured or duplicate SPNs are one of the most common causes of mysterious Kerberos authentication failures.
    - **setspn -L \<account\>** = lists all SPNs currently registered on a specific account.
    - **setspn -Q \<SPN\>** = queries the entire domain/forest for a specific SPN string, used to check for duplicates before assigning a new one (a duplicate SPN across two different accounts breaks Kerberos auth for both).
    - **setspn -S \<SPN\> \<account\>** = adds an SPN to an account, but only after first checking for and preventing a duplicate (the safer version of `-A`).
    - **setspn -A \<SPN\> \<account\>** = adds an SPN without the duplicate check (legacy, less safe).
- **ntdsutil** = a powerful, low-level, interactive command-line utility for maintaining the Active Directory database (NTDS.dit) directly — used for tasks like seizing FSMO roles from a failed domain controller, performing database integrity checks/defragmentation, and resetting the Directory Services Restore Mode (DSRM) password. Because it operates directly on the AD database, mistakes here can be severe — this is squarely an 'know what you're doing before you run it' tool.
    - **ntdsutil "roles" "connections" "connect to server \<DC\>" q "seize pdc" q q** = an example of the nested command syntax used to seize the PDC Emulator FSMO role from a DC that's permanently offline.
    - **ntdsutil "set dsrm password" "reset password on server null" q q** = resets the Directory Services Restore Mode local administrator password, required before being able to boot a DC into DSRM for database maintenance.
- **findstr** = the cmd.exe equivalent of Linux's `grep` — searches for text patterns within files, supporting a limited regex-like syntax of its own (not full POSIX/PCRE regex).
    - **/i** = case-insensitive search.
    - **/s** = searches recursively through subdirectories.
    - **/n** = includes line numbers in the output.
    - **/v** = inverts the match, showing lines that do NOT match (like `grep -v`).
- **fc** = File Compare — the classic cmd.exe utility for comparing two files and displaying their differences, conceptually the direct predecessor to Linux's `diff`, though with a much more limited/dated output format.
    - **/n** = displays line numbers alongside differences.
    - **/b** = performs a binary comparison instead of a text/line comparison.

- **attrib** = displays or changes file attributes (Read-only, Hidden, System, Archive) — the cmd.exe equivalent in purpose to `chattr` on Linux, though the actual attribute flags themselves are conceptually different (these are basic DOS-era file attributes, not the more advanced filesystem-level protections chattr provides).
    - **+r / -r** = sets or clears the Read-only attribute (similarly `+h`/`-h` for Hidden, `+s`/`-s` for System, `+a`/`-a` for Archive).
    - **/s /d** = applies the change recursively to files and directories in subfolders as well.
- **subst** = creates a virtual drive letter mapped directly to a local folder path — useful for shortening long, deeply-nested folder paths in a script or for legacy applications that can't handle paths beyond a certain length. Purely a local, session-based mapping (not persistent across reboots by default, and entirely different in purpose from mapping a network drive).
- **fsutil** = an advanced, low-level file system utility exposing operations not available through any GUI — querying volume/NTFS metadata, managing hard links, checking disk quota usage, and manipulating sparse files.
    - **fsutil fsinfo drives** = lists all drive letters currently mapped on the system.
    - **fsutil fsinfo ntfsinfo \<drive\>** = shows detailed NTFS volume metadata (cluster size, MFT location, serial number).
    - **fsutil hardlink create \<link\> \<target\>** = creates an NTFS hard link, the Windows equivalent of Linux's `ln` (without `-s`).
    - **fsutil behavior query DisableLastAccess** = checks whether NTFS is tracking file last-access timestamps — the Windows equivalent concept to a Linux mount's `noatime` option.
- **w32tm** = Windows Time service command-line tool — configures and diagnoses NTP time synchronization, the Windows equivalent in purpose to Linux's `chronyc`/`timedatectl`.
    - **/query /status** = shows the current time source and synchronization status.
    - **/resync** = forces an immediate time resynchronization instead of waiting for the normal interval.
    - **/config /manualpeerlist:\<servers\> /syncfromflags:manual /update** = configures a specific external NTP source (commonly done on the PDC Emulator, which is the authoritative time source for the rest of the domain).
- **auditpol** = queries and configures the Windows Advanced Audit Policy — controlling exactly which security events (logon attempts, object access, privilege use, policy changes) get written to the Security event log. Far more granular than the older, basic Local Security Policy audit categories.
    - **/get /category:*** = displays the current audit policy configuration for every category.
    - **/set /subcategory:"Logon" /success:enable /failure:enable** = enables auditing for both successful and failed logon events specifically.
- **secedit** = applies, analyzes, and exports security policy templates (.inf files) — used to bulk-apply a standardized security baseline across many servers, or to compare a server's current configuration against an expected security template.
    - **/export /cfg \<file\>** = exports the current local security policy to a template file.
    - **/configure /db \<file\> /cfg \<template\>** = applies a security template to the local machine.
    - **/analyze /db \<file\> /cfg \<template\>** = compares the current configuration against a template without making changes, flagging deviations.
- **driverquery** = lists all installed device drivers on the local (or a remote) machine, along with their type, state, and link date — a quick way to audit for outdated or unsigned drivers across a fleet without opening Device Manager on every machine.
    - **/v** = verbose output, including driver file path.
    - **/si** = shows signature verification information for each driver.
- **msinfo32** = System Information — launches a GUI tool showing an extremely detailed hardware/software/driver inventory of the local machine, but is also callable from the command line to generate a report file

non-interactively (`msinfo32 /report C:\report.txt` or `/nfo` for the binary .nfo format readable back in the GUI) — useful for attaching a full system snapshot to a support ticket.

- **net share / net use / net view** = the legacy command-line family for managing SMB file shares and network connections — largely predating the `SmbShare` PowerShell module but still extremely common in scripts and quick troubleshooting.
    - **net share** = lists all shares currently published from the local machine (or use `net share \<name\>=\<path\>` to create a new one).
    - **net use \<drive:\> \\server\share** = maps a network drive letter to a remote SMB share, optionally with `/user:` and `/persistent:yes` for saved credentials and persistence across reboots.
    - **net view \\server** = lists the shares available on a specific remote server.
- **cmdkey** = manages stored/cached credentials on the local machine (the same credential store viewed through Control Panel's Credential Manager) — useful for scripting credential management for scheduled tasks or persistent network share connections that need to authenticate without a live interactive prompt.
    - **/add:\<target\> /user:\<user\> /pass:\<pass\>** = stores a new credential.
    - **/list** = lists all currently stored credentials.
    - **/delete:\<target\>** = removes a stored credential.
- **wecutil** = Windows Event Collector utility — configures and manages Windows Event Forwarding (WEF) subscriptions, which allow a central 'collector' server to pull event logs from many remote source machines automatically. A core building block for centralized security log monitoring without needing a third-party SIEM agent on every endpoint.
    - **es** (enum-subscriptions) = lists configured event forwarding subscriptions.
    - **cs \<file.xml\>** (create-subscription) = creates a new subscription from an XML definition file.
    - **gr \<subscription\>** (get-subscription-runtime-status) = shows the current status and any errors for a specific subscription.
- **forfiles** = a batch-friendly tool for selecting a set of files based on age/date/pattern criteria and running a command against each one — commonly used in scheduled log cleanup scripts, though PowerShell's `Get-ChildItem | Where-Object` combination is generally the more flexible and readable modern replacement.

```
forfiles /p "C:\Logs" /s /m *.log /d -30 /c "cmd /c del @path"
```

    - **/d -30** = matches files last modified more than 30 days ago (a negative number means 'older than'; positive means 'newer than').
    - **/c "\<command\>"** = the command to execute against each matched file, where `@path`/`@file` are substituted with the matched file's path/name.
- **cacls** = the original, now-deprecated legacy tool for viewing/editing NTFS permissions, predating `icacls` (which added support for inheritance flags and is the recommended modern tool). Still occasionally seen in very old scripts, but `icacls` should be used for anything new.
- **cipher /w and file system encryption status** = beyond the secure-wipe usage already covered above, running bare `cipher` in a directory also displays the encryption status (E for encrypted, U for unencrypted) of every file, a quick way to audit which files in a folder have EFS encryption applied.
- **dir** = lists the files and subdirectories inside a directory — the cmd.exe equivalent of Linux's `ls`. (ex. `dir /a /s C:\Logs` lists everything recursively, including hidden files)
- **cd / chdir** = displays or changes the current working directory. Use `cd \` to jump to the drive root, or `cd /d D:\Data` to change drive and directory in one step.
- **copy** = copies one or more files to another location — the single-file counterpart to `xcopy`/`robocopy`. (ex. `copy report.txt D:\Backup\`)
- **move** = moves (or renames, if the destination is on the same drive) a file or directory.

- **del / erase** = deletes one or more files. `del /f /q *.tmp` forces deletion of read-only files without a confirmation prompt.
- **ren / rename** = renames a file or directory without needing to specify its full path again.
- **md / mkdir and rd / rmdir** = create and remove directories, respectively. `rd /s /q <folder>` recursively deletes a folder and everything inside it without prompting.
- **type** = prints the contents of a text file to the console — the direct cmd.exe equivalent of Linux's `cat`. (ex. `type C:\Logs\app.log`)
- **cls** = clears the console screen. The cmd.exe equivalent of Linux's `clear`.
- **echo** = prints text to the console, or (inside a batch file) toggles whether commands are printed as they execute (`echo off`).
- **set / setx** = manage environment variables from cmd.exe.
    - **set** = views or sets an environment variable for the current cmd.exe session only; the change disappears once that window closes.
    - **setx** = permanently writes an environment variable to the registry so it persists across sessions and reboots and is visible to new processes (but not the current window). (ex. `setx JAVA_HOME "C:\Program Files\Java\jdk17"`)
- **path** = displays or sets the current session's `PATH` environment variable directly (a shorthand alternative to `set PATH=...`).
- **start** = launches a program, document, or URL in a new window, optionally without waiting for it to finish. (ex. `start notepad.exe` or `start https://contoso.com`)
- **call** = invokes another batch file from within a batch file, then returns control to the calling script afterward (running a batch file without `call` transfers control permanently and never returns).
- **exit / pause** = `exit` closes the current cmd.exe window (or `exit /b` returns from a batch file without closing the window); `pause` halts a batch script and waits for a keypress, commonly used for troubleshooting a script that closes too fast to read.
- **more** = pipes output one screen at a time, similar to Linux's `more`. (ex. `type biglog.txt | more`)
- **tree** = displays a graphical, indented tree of a directory's folder structure. Add `/f` to also list the files in each folder.
- **comp** = compares the contents of two files byte-by-byte and reports the first difference found — a lighter-weight alternative to `fc` when you only need a yes/no answer.
- **doskey** = manages command-line macros and history for cmd.exe sessions, letting you define a shortcut alias for a longer command (ex. `doskey ll=dir /a $*`).
- **assoc / ftype** = view or change file-extension associations. `assoc` maps an extension to a file type (ex. `assoc .log`), and `ftype` maps that file type to the program used to open it.
- **vol / label** = `vol` displays a drive's volume label and serial number; `label` changes the volume label of a drive.
- **format** = formats a disk or partition with a specified filesystem, erasing all data on it. (ex. `format D: /fs:NTFS /q` performs a quick format)
- **convert** = non-destructively converts a FAT32 volume to NTFS in place, without erasing existing data. (ex. `convert D: /fs:ntfs`)
- **defrag** = analyzes or defragments a disk's file layout to improve read/write performance on traditional HDDs. (ex. `defrag C: /A` analyzes without defragmenting)
- **runas** = runs a specific program under a different user account's credentials without logging out — the everyday way to launch an elevated or alternate-account session from a standard prompt. (ex. `runas /user:CONTOSO\admin cmd`)
- **quser / qwinsta and logoff** = `quser`/`qwinsta` list the users currently logged into a machine (locally or via RDP) along with their session IDs; `logoff <session ID>` forcibly ends one of those sessions.

- **tzutil** = displays or sets the operating system's time zone from the command line. (ex. `tzutil /g` shows the current zone; `tzutil /s "Eastern Standard Time"` sets it)
- **nbtstat** = displays NetBIOS-over-TCP/IP statistics and name-resolution cache entries, useful for troubleshooting legacy name-resolution issues on a Windows LAN.
- **getmac** = displays the MAC (physical) address of every network adapter on the local machine.
- **wbadmin** = the command-line interface for Windows Server Backup — start, manage, or restore from a full system/volume backup without touching the GUI console. (ex. `wbadmin start backup -backupTarget:D: -include:C:`)
- **bootrec** = run from Windows Recovery Environment to repair a corrupted boot configuration. `bootrec /fixmbr` rewrites the master boot record; `bootrec /rebuildbcd` rescans for installed Windows installations and rebuilds the boot menu.
- **typeperf / logman** = command-line access to Windows Performance Counters without opening the Performance Monitor GUI.
    - **typeperf** = prints live performance counter values to the console or a CSV file. (ex. `typeperf "\Processor(_Total)\% Processor Time"`)
    - **logman** = creates and manages scheduled Performance Monitor data collector sets that run unattended over time.
- **net start / net stop / net accounts** = `net start`/`net stop` start or stop a Windows service by name — a simpler, more memorable alternative to `sc start`/`sc stop` for everyday use; `net accounts` displays or configures the local password/lockout policy.
- **hostname** = prints the local computer's hostname — the direct equivalent of Linux's `hostname` with no arguments.
- **ver** = prints the Windows OS version string for the current session.
- **winget** = the Windows Package Manager, Microsoft's native CLI for finding, installing, upgrading, and removing software, comparable to `apt`/`dnf` on Linux.
    - **winget search <name>** = searches configured sources for a package matching the name.
    - **winget install <id>** = downloads and silently installs a package. (ex. `winget install Git.Git`)
    - **winget upgrade --all** = upgrades every installed package that has a newer version available.
    - **winget uninstall <id>** = removes an installed package.

PowerShell Scripting

These are the core language building blocks of PowerShell scripting: syntax, variables, operators, control flow, functions, and error handling. Just like the Linux shell scripting section, understanding these fundamentals is what lets you actually read and write real automation, not just paste one-liners you found online.

- **function Verb-Noun { }** = the syntax for creating a function in PowerShell. Functions should be named using an approved PowerShell Verb (Get, Set, New, Remove, etc.) followed by a singular Noun, separated by a hyphen. This Verb-Noun convention is enforced by convention (not the parser) and is why every native cmdlet looks like Get-Process or Set-Item.

```
function Get-DiskSpace {
    param(
        [string]$ComputerName = $env:COMPUTERNAME
    )
    Get-CimInstance Win32_LogicalDisk -ComputerName $ComputerName
}
```

    - **param()** = declares the input parameters for the function, placed as the first statement inside the function body (or in a `param` block immediately following an optional `[CmdletBinding()]` attribute).

    - **Get-Verb** = lists every approved PowerShell verb (Get, Set, New, Remove, Add, Clear, Test, Invoke, etc.) grouped by verb category (Common, Data, Lifecycle, Diagnostic, Security). Using unapproved verbs triggers a warning when the module is imported.
    - Unlike Bash, PowerShell functions declare named parameters directly in the signature rather than relying on positional `$1`, `$2` style arguments — though positional binding still works if you call the function without parameter names in the same order they were declared.
- **#** = (Comment): Used for single-line comments. Everything after a # on a line is ignored by the parser. PowerShell does not use a shebang line the way Bash does (`#!/bin/bash`) — script type is determined by the .ps1 extension itself, not a header line.
    - **<# #>** = block comment delimiters. Everything between `<#` and `#>` is ignored, even across multiple lines. This is also the required syntax for comment-based help (`.SYNOPSIS`, `.DESCRIPTION`, `.PARAMETER`, `.EXAMPLE`) placed at the top of a function or script so `Get-Help` can read it.
- **$** = (Variable Sigil): Every PowerShell variable name is prefixed with a dollar sign both when you declare it and when you reference it (unlike Bash, where `$` is only used to read a variable, not to assign one). Variable names are case-insensitive.

```
$ServerName = "LAB-DC01"
Write-Output $ServerName
```

    - **${ }** = curly-brace variable notation, used to reference a variable name that contains spaces or special characters, or to disambiguate the variable name from adjacent text (similar in purpose to Bash's `${var}`). Example: `${my variable}` or `"${PREFIX}_log"`.
    - **$()** = the subexpression operator. Forces PowerShell to evaluate everything inside the parentheses as an expression first and insert the result — required whenever you need to embed a property access, method call, or command result inside a string. Example: `"There are $($services.Count) services running."`
    - **@()** = the array subexpression operator. Forces the result of the enclosed expression to always be returned as an array, even if it only returns zero or one object. Extremely useful when a cmdlet might return a single object instead of a collection, which would otherwise break `.Count` or a `foreach` loop downstream.
- **$_ or $PSItem** = the automatic variable representing the current object in the pipeline. Used almost exclusively inside script blocks passed to `Where-Object`, `ForEach-Object`, `Sort-Object`, and similar pipeline cmdlets. `$_` and `$PSItem` are fully interchangeable; `$PSItem` is simply the more readable alias introduced later.

```
Get-Process | Where-Object { $_.CPU -gt 100 }
Get-Service | ForEach-Object { "$($_.Name) is $($_.Status)" }
```

- **$? and $LASTEXITCODE** = automatic variables used to check success/failure, similar to Bash's `$?` exit status.
    - **$?** = a Boolean ($true or $false) reflecting whether the last command completed successfully (from PowerShell's own perspective, based on whether it wrote to the error stream).
    - **$LASTEXITCODE** = holds the actual numeric exit code of the last *native* executable (like `robocopy.exe` or `ping.exe`) that was run — this is the direct equivalent of Bash's `$?` and is what you should check after calling external .exe tools, since $? only reflects PowerShell's own cmdlet success.
    - **$Error** = an automatic array variable holding the most recent error objects PowerShell has generated in the session, with `$Error[0]` being the very last error thrown.
- **= vs -eq** = the single most common syntax trip-up for people coming from other shells or languages. In PowerShell, `=` is exclusively the assignment operator, never a comparison.
    - **=** = assigns a value to a variable (ex. `$Role = "Admin"`). Never used for comparison.
    - **-eq** = the comparison operator meaning 'equal to' (ex. `if ($Role -eq "Admin")`). PowerShell comparison operators are always words prefixed with a hyphen (`-eq`, `-ne`, `-gt`), never symbols like `==` from other languages — though `-eq` will still work as a case-insensitive string OR numeric comparison depending on the operand types.

- **Comparison Operators** = used inside `if`, `while`, and `Where-Object` conditions. All comparison operators are case-insensitive strings by default unless explicitly marked case-sensitive with a `c` prefix.
  - **-eq / -ceq** = equal to (case-insensitive / case-sensitive explicitly)
  - **-ne / -cne** = not equal to
  - **-gt / -clt** = greater than
  - **-ge** = greater than or equal to
  - **-lt** = less than
  - **-le** = less than or equal to
  - **-like / -notlike** = wildcard string comparison using `*` and `?` (ex. `$name -like "srv*"`)
  - **-match / -notmatch** = regular expression comparison. On a match, automatically populates the `$Matches` automatic hashtable with the matched groups (ex. `if ($ip -match '^\d{1,3}(\.\d{1,3}){3}$')`)
  - **-contains / -notcontains** = tests whether a **collection** contains a specific value (ex. `$Array -contains "admin"`). This is the reverse direction of `-in`.
  - **-in / -notin** = tests whether a single value exists **inside** a collection (ex. `"admin" -in $Array`). Functionally the mirror image of `-contains`.
  - **-is / -isnot** = type comparison, tests whether an object is a specific .NET type (ex. `$Value -is [int]`)
- **Logical Operators** = combine multiple Boolean conditions inside an `if` statement or filter script block.
  - **-and** = both conditions must be true
  - **-or** = at least one condition must be true
  - **-not or !** = negates a Boolean value (ex. `if (-not $IsAdmin)` or `if (!$IsAdmin)`)
  - **-xor** = exclusive or; true only if exactly one of the two conditions is true
- **if / elseif / else** = controls execution flow based on whether a condition evaluates to $true or $false. Unlike Bash, PowerShell conditions do not rely on numeric exit codes (0 = success) — they evaluate actual Boolean expressions, and curly braces `{ }` are mandatory instead of `then`/`fi` keywords.

```
if ($DiskSpaceGB -lt 10) {
    Write-Warning "Low disk space"
} elseif ($DiskSpaceGB -lt 50) {
    Write-Output "Disk space getting low"
} else {
    Write-Output "Disk space OK"
}
```

- **switch** = provides clean multi-way branching against a single value, replacing long `if / elseif` chains — directly analogous to Bash's `case` statement. PowerShell's switch is more powerful than most other languages' switch statements because each condition can itself be a wildcard, regex, or script block.

```
switch ($StatusCode) {
    200 { "OK" }
    404 { "Not Found" }
    { $_ -ge 500 } { "Server Error" }
    default { "Unknown" }
}
```

  - **-Wildcard** = evaluates each case using wildcard matching instead of exact matching (ex. `switch -Wildcard ($name) { "srv*" { "server" } }`)
  - **-Regex** = evaluates each case as a regular expression pattern instead of an exact match.
  - **-CaseSensitive** = forces exact-case matching for string comparisons (default is case-insensitive).
  - **default** = the fallback block, equivalent to Bash's `*)` wildcard case — runs only if no other case matched.

    - Unlike Bash's `case`, PowerShell's `switch` does **not** stop after the first match by default — every matching case block executes unless you explicitly add a `break` statement inside the block.

- **foreach** = loops over every item in a collection (array, list of objects, etc.), running the same block of code for each one. There are two distinct 'foreach' mechanisms in PowerShell that behave very differently and are frequently confused.

```
foreach ($server in $ServerList) {
    Test-Connection -ComputerName $server -Count 1
}
```

    - **foreach (keyword/statement)** = a language keyword that loads the *entire* collection into memory before looping. Faster for small-to-medium in-memory collections, but cannot receive pipeline input directly.
    - **ForEach-Object (cmdlet, alias %)** = a pipeline cmdlet that processes objects **one at a time as they arrive** in the pipeline, without holding the whole collection in memory first. This is the correct choice when piping from something like `Get-ChildItem` against a huge directory, since it starts working immediately instead of waiting to enumerate everything first. Example: `Get-Process | ForEach-Object { $_.Name }`
    - **.ForEach() method** = a method available on any collection/array object (ex. `$Array.ForEach({ $_.ToUpper() })`), fast like the `foreach` keyword but usable inline without an explicit loop block.

- **for** = a classic, C-style counting loop with an initializer, a condition, and an increment/decrement statement — used when you need explicit control over the counter (e.g., looping a fixed number of times, or stepping by something other than 1).

```
for ($i = 1; $i -le 5; $i++) {
    Write-Output "Attempt $i"
}
```

- **while** = repeats a block of code for as long as a condition remains $true, checking the condition **before** each iteration (so the loop body may never execute at all if the condition starts out false).

```
while ($RetryCount -lt 3 -and -not $Success) {
    $Success = Test-Connection -ComputerName $Server -Count 1 -Quiet
    $RetryCount++
}
```

- **do { } while ( )** = the inverse of `while` — it runs the loop body **first**, and only then checks the condition, guaranteeing the block always executes at least once regardless of the condition.

```
do {
    $Response = Read-Host "Continue? (Y/N)"
} while ($Response -ne "Y" -and $Response -ne "N")
```

- **do { } until ( )** = functionally the mirror image of `do-while` — it also guarantees at least one execution, but loops **until** the condition becomes $true (rather than looping while it is true). Choosing between them is purely a matter of which reads more naturally for the logic you're expressing.

```
do {
    Start-Sleep -Seconds 5
    $Job = Get-Job -Id $JobId
} until ($Job.State -eq "Completed")
```

- **break / continue** = loop control statements identical in purpose to their counterparts in Bash and most other languages.
    - **break** = immediately exits the entire loop (or `switch` block) it is inside, skipping any remaining iterations.
    - **continue** = skips the rest of the current iteration only and immediately jumps to the next iteration's condition check.

- **Arrays @()** = an ordered, zero-indexed collection of values, declared with the `@()` array subexpression operator (though a simple comma-separated list is automatically treated as an array too).

```
$Servers = @("LAB-DC01", "LAB-DC02", "LAB-FILE01")
$Servers[0]        # LAB-DC01
$Servers += "LAB-DC03" # appends a new element (creates a new array under the hood)
$Servers.Count     # 4
```

  - Arrays in PowerShell are technically fixed-size .NET arrays — using `+=` doesn't actually append in place, it silently creates an entirely new array behind the scenes. For frequent additions in a loop, use an `[System.Collections.Generic.List[object]]` or `ArrayList` instead for performance.
  - **$Array[-1]** = negative indexing returns items counting from the end of the array (`-1` is the last element, `-2` is second-to-last, etc.).
  - **$Array[1..3]** = the range operator (`..`) returns a slice of the array from index 1 through index 3, inclusive.
- **Hashtables @{ }** = an unordered collection of key/value pairs, declared with `@{ }` — the PowerShell equivalent of a dictionary or associative array. Extremely common for building parameter sets, configuration data, and lookup tables.

```
$UserInfo = @{
    Name       = "asample"
    Department = "IT"
    Enabled    = $true
}
$UserInfo.Name        # dot notation access
$UserInfo["Name"]     # bracket notation access
$UserInfo.Add("Title", "Admin")
```

  - **[ordered]@{ }** = a special cast that preserves the insertion order of keys (a normal hashtable's key order is not guaranteed). Frequently used when the display order of properties matters, e.g., before converting to a custom object.
  - **.Keys / .Values** = properties that return just the key names or just the values as separate collections.
  - **.Remove("key")** = removes a specific key/value pair from the hashtable.
- **Splatting @Params** = a technique for passing a hashtable (or array) of parameters to a cmdlet using the `@` sigil instead of the normal `$` sigil, which expands each key/value pair into an individual `-ParameterName Value` argument. This dramatically improves readability for cmdlets with many parameters and is the standard professional way to write reusable, maintainable scripts.

```
$Params = @{
    Path        = "C:\Logs"
    Filter      = "*.log"
    Recurse     = $true
    ErrorAction = "SilentlyContinue"
}
Get-ChildItem @Params
```

- **"" (Double Quotes)** = (Interpolating String): variables, subexpressions `$()`, and escape sequences inside double quotes are automatically expanded/evaluated.
  - **'' (Single Quotes)** = (Literal String): Everything inside single quotes is treated as literal text — variables are **not** expanded. Functionally the inverse of Bash, where single quotes are the strict/literal ones and double quotes allow expansion (PowerShell follows the same convention, just confirming it explicitly).
  - **`` ` `` (Backtick, Escape Character)** = PowerShell's escape character (used instead of Bash's backslash `\`). Common escape sequences: `` `n `` (newline), `` `t `` (tab), `` `" `` (literal double quote), `` ` ` `` (a literal backtick itself).
  - A backtick at the very **end of a line** is also PowerShell's line-continuation character, letting a single logical command span multiple physical lines — though this is fragile (a trailing space after the backtick silently breaks it) and splatting or pipelines are generally the preferred alternative.

- **Here-Strings @" "@** = PowerShell's equivalent of Bash's here-document, used to embed a large, multi-line block of text (often JSON, XML, or a formatted report) directly in a script without needing to escape quotes line by line. The opening `@"` and closing `"@` must each be the only thing on their own line, and the closing `"@` must start in column 1.

```
$Body = @"
Server: $ServerName
Status: $Status
Checked: $(Get-Date)
"@
```

    - **@" ... "@** = an *expanding* here-string — variables and subexpressions inside it are evaluated, just like a normal double-quoted string.
    - **@' ... '@** = a *literal* here-string using single quotes — nothing inside is expanded, matching the literal behavior of single-quoted strings.
- **-f (Format Operator)** = the string formatting operator, used to insert values into a template string using numbered placeholders `{0}`, `{1}`, etc. — directly analogous to `printf` in Bash, but placeholder-based rather than type-flag-based.

```
"{0} is currently {1}" -f $ServiceName, $Status
```

- **Pipe |** = the pipe operator, functionally similar to Bash's pipe but fundamentally different under the hood: PowerShell pipes pass whole **.NET objects** (with all their properties and methods intact) from one command to the next, not plain text. This is why you can do `Get-Process | Sort-Object CPU` without any text parsing at all — Bash pipes only pass raw text streams.

```
Get-Service | Where-Object { $_.Status -eq "Stopped" } | Sort-Object Name
```

- **Where-Object** = filters objects in the pipeline down to only those matching a condition — the PowerShell equivalent of piping through `grep`, but operating on structured object properties instead of raw text lines.
    - Full script block syntax: `Where-Object { $_.Status -eq "Running" }`
    - Simplified comparison syntax (PowerShell 3.0+): `Where-Object Status -eq "Running"` (skips the need for `$_` entirely for simple single-condition filters)
- **ForEach-Object** = see the `foreach` entry above — as a reminder, this is the pipeline **cmdlet** form (alias `%`), which processes each pipeline object individually as it streams through, as opposed to the `foreach` **keyword**, which loads a collection entirely into memory first.
- **Select-Object** = shapes pipeline output by choosing specific properties to display/pass along, limiting the number of results, or creating brand-new calculated properties.
    - **-Property** = selects only the named properties to output (ex. `Get-Process | Select-Object Name, CPU, Id`)
    - **-First / -Last** = returns only the first or last N objects from the pipeline.
    - **-Unique** = removes duplicate objects from the output.
    - **-ExpandProperty** = unwraps/flattens a single property so its raw value is returned instead of a wrapping object (ex. `Select-Object -ExpandProperty Name`).
    - **Calculated properties** = a hashtable with `Name`/`Label` and `Expression` keys lets you create a computed column on the fly, ex. `Select-Object Name, @{Name="SizeMB"; Expression={$_.Length / 1MB}}`
- **Sort-Object** = sorts pipeline objects by one or more properties.
    - **-Property** = the property (or properties) to sort by (ex. `Sort-Object -Property CPU`)
    - **-Descending** = reverses the sort order (default is ascending).
    - **-Unique** = removes duplicates from the sorted output.
- **Group-Object** = groups pipeline objects together based on a shared property value, similar in spirit to a SQL `GROUP BY` (ex. `Get-Process | Group-Object -Property Company`).

    - **-NoElement** = suppresses the list of grouped objects in the output, showing only the group names and counts.
- **Measure-Object** = calculates numeric statistics (sum, average, minimum, maximum, count) or basic text statistics (lines, words, characters) across a collection of objects — the PowerShell equivalent of a combination of Bash's `wc` and manual arithmetic.
    - **-Sum / -Average / -Maximum / -Minimum** = specifies which statistic(s) to calculate on a numeric property (ex. `Get-Process | Measure-Object WorkingSet -Sum`)
    - **-Line / -Word / -Character** = calculates text statistics instead, when piped raw text content (ex. `Get-Content file.txt | Measure-Object -Line -Word -Character`)
- **Compare-Object** = compares two collections and reports the differences between them — analogous to the Linux `diff` command, but operating on objects/collections rather than raw text files.
    - **-ReferenceObject / -DifferenceObject** = the two collections being compared.
    - **-IncludeEqual** = also shows items present in *both* collections, not just the differences.
    - **SideIndicator** = the output column showing `<=` (only in reference), `=>` (only in difference), or `==` (in both, if `-IncludeEqual` was used).
- **Tee-Object** = reads pipeline input, passes it through unchanged to the next command, and simultaneously writes a copy to a file or variable — the direct equivalent of the Linux `tee` command.

```
Get-Process | Tee-Object -FilePath C:\Logs\procs.txt | Where-Object CPU -gt 100
```

- **Try / Catch / Finally** = PowerShell's structured error-handling block, used to gracefully catch and respond to *terminating* errors instead of letting a script crash outright — this is the modern replacement for checking `$?` after every single command.

```
try {
    Get-Item "C:\DoesNotExist" -ErrorAction Stop
} catch [System.Management.Automation.ItemNotFoundException] {
    Write-Warning "File not found: $($_.Exception.Message)"
} catch {
    Write-Warning "Unexpected error: $($_.Exception.Message)"
} finally {
    Write-Output "Cleanup runs no matter what happened above."
}
```

    - **catch [SpecificException]** = you can chain multiple `catch` blocks, each targeting a specific .NET exception type, evaluated top to bottom (similar to `elif`).
    - **finally** = a block that always executes, whether or not an error occurred and whether or not it was caught — used for cleanup like closing a file handle or disconnecting a session.
    - **Critical catch**: `try/catch` only catches **terminating** errors. Many cmdlets produce **non-terminating** errors by default (they print a red error message but let the script continue). You must add `-ErrorAction Stop` to the command inside the `try` block to force it to become a terminating error that `catch` can actually intercept.
- **-ErrorAction** = a common parameter available on virtually every cmdlet that controls how that specific command responds when it encounters an error, overriding the session-wide `$ErrorActionPreference` default just for that one call.
    - **Stop** = converts a non-terminating error into a terminating one, which is required for `try/catch` to be able to intercept it.
    - **SilentlyContinue** = suppresses the error message entirely and continues execution (does not add the error to `$Error` unless `-ErrorVariable` is also used).
    - **Continue** = (the default) displays the error message but continues execution.
    - **Ignore** = suppresses the error and does **not** add it to the `$Error` automatic variable at all (unlike SilentlyContinue).

    - **Inquire** = prompts the user interactively to decide whether to continue.
- **throw** = manually raises a terminating error, immediately halting the current scope unless it's caught by an enclosing `try/catch`. Used inside your own functions to signal a genuine failure condition to the caller.

```
if (-not (Test-Path $ConfigPath)) { throw "Configuration file not found at
$ConfigPath" }
```

- **$ErrorActionPreference** = a global/scoped variable that sets the *default* error-handling behavior for the entire script or session when a command doesn't explicitly specify its own `-ErrorAction`. Common practice is to set this to `"Stop"` at the top of a script so that any unexpected error immediately halts execution rather than silently continuing.
- **Scope (Global / Script / Local / Private)** = determines where a variable or function is visible from, directly analogous to Bash's distinction between normal shell variables, function-local variables (`local`), and exported/global variables (`export`).
    - **Local (default)** = a variable defined inside a function or script block is only visible within that block and its children, unless explicitly scoped otherwise — this is PowerShell's default behavior, unlike Bash where variables leak into the parent shell unless declared `local`.
    - **$global:VarName** = explicitly places (or accesses) a variable in the global scope, making it visible everywhere in the current PowerShell session — the closest equivalent to Bash's `export`.
    - **$script:VarName** = scopes a variable to the entire script file it's defined in, visible to all functions within that script but not outside it.
    - **$using:VarName** = a special scope modifier required to pass a *local* variable's value into a remote session or a background job's script block (ex. `Invoke-Command -ScriptBlock { Get-Service $using:ServiceName }`).
- **Regex operators (-match, -replace, -split)** = PowerShell has native regular expression support built directly into several operators, backed by the full .NET regex engine (more powerful than POSIX ERE used by Bash's `=~`).
    - **-match / -notmatch** = tests a string against a regex pattern; populates the `$Matches` hashtable on success.
    - **-replace** = performs a regex find-and-replace on a string (ex. `"192.0.2.50" -replace '\.\d+$', '.0'`). Unlike `.Replace()`, this always treats the search pattern as regex.
    - **-split** = splits a string into an array using a regex pattern as the delimiter (ex. `"a,b,,c" -split ','`).
- **Import-Module / Get-Module** = PowerShell's system for loading external command libraries — the direct equivalent of Bash's `source` for pulling in reusable functions, except modules are structured, versioned, and can contain compiled .NET code, not just script text.
    - **Import-Module \<name\>** = loads a module's cmdlets/functions into the current session (ex. `Import-Module ActiveDirectory`).
    - **Get-Module** = lists modules currently loaded in the session. Add `-ListAvailable` to see every module installed on disk, whether loaded yet or not.
    - **Remove-Module** = unloads a module from the current session without uninstalling it from disk.
    - **-Force** = forces a module to reload even if it's already imported (useful while actively developing/editing a module's code).
- **Install-Module / Update-Module** = manage PowerShell modules published to a repository (typically the PowerShell Gallery, PSGallery) — the PowerShell equivalent of a package manager like `dnf`/`apt`, specifically for PowerShell code libraries such as `Az`, `Microsoft.Graph`, or `ExchangeOnlineManagement`.
    - **Install-Module -Name \<name\> -Scope CurrentUser** = downloads and installs a module from the configured repository. `-Scope CurrentUser` avoids needing admin rights by installing to the user's own module path instead of the system-wide one.
    - **-Force** = suppresses confirmation prompts and allows installing over an existing version.
    - **-AllowClobber** = allows the install to proceed even if the module contains cmdlets with the same name as ones already loaded from another module.

    - **Update-Module** = updates an already-installed module to the latest version available in the repository.
    - **Uninstall-Module** = removes an installed module from disk entirely.
    - **Find-Module** = searches the configured repository (PSGallery) for available modules matching a name, without installing anything.
    - **Register-PSRepository / Get-PSRepository** = manages which module repositories PowerShell is allowed to pull from.
- **Get-Help** = PowerShell's built-in manual system — the direct equivalent of Linux's `man` command, pulling documentation directly from a cmdlet's comment-based help or externally shipped help XML.
    - **-Full** = shows the complete help topic including parameter descriptions, all examples, and notes.
    - **-Examples** = shows only the usage examples for the cmdlet.
    - **-Online** = opens the cmdlet's official online documentation page in a web browser instead of showing console text.
    - **-Detailed** = shows parameter descriptions and examples, but not the full technical parameter attribute tables that `-Full` includes.
    - **Update-Help** = downloads the latest help content files from the internet (required once after installing PowerShell, since help files don't ship inline by default anymore).
- **Get-Command** = searches for available cmdlets, functions, aliases, and native executables — useful when you know roughly what you want to do but don't remember the exact cmdlet name.
    - **-Noun \<name\>** = finds all cmdlets that operate on a specific noun (ex. `Get-Command -Noun Service` finds Get-Service, Set-Service, Start-Service, Stop-Service, Restart-Service, etc.)
    - **-Verb \<name\>** = finds all cmdlets using a specific verb (ex. `Get-Command -Verb Get`).
    - **-Module \<name\>** = lists all commands contained within a specific module.
- **Get-Member** = reveals the actual .NET type of an object flowing through the pipeline, along with every property and method it exposes. This is the single most important discovery tool in PowerShell — piping *anything* into `Get-Member` (alias `gm`) tells you exactly what you can do with it.

```
Get-Process | Get-Member
```

    - **-MemberType** = filters the output to a specific kind of member (ex. `-MemberType Method` shows only callable methods, `-MemberType Property` shows only properties).
- **ConvertTo-Json / ConvertFrom-Json** = converts PowerShell objects to and from JSON text — critical for talking to REST APIs (like Microsoft Graph) and for saving structured data to disk in a portable format.
    - **-Depth \<n\>** = specifies how many levels of nested objects to serialize (default is only 2, which silently truncates deeply nested objects — a very common gotcha when converting complex API responses).
- **Export-Csv / Import-Csv** = writes pipeline objects to a CSV file, or reads a CSV file back in as an array of custom objects — the standard way to hand off structured data to Excel or another script/system.
    - **-NoTypeInformation** = suppresses the `#TYPE` header line that older PowerShell versions added by default at the top of the CSV (no longer needed by default in PowerShell 7+, but still common to see in older scripts).
    - **-Append** = adds rows to an existing CSV file instead of overwriting it.
    - **-Delimiter** = specifies a different field separator (default is a comma).
- **ConvertTo-SecureString / Get-Credential** = the standard mechanisms for handling passwords and credentials without exposing them as plain text in a script.
    - **Get-Credential** = interactively prompts the user for a username and password, returning a `PSCredential` object that most remoting/authentication cmdlets accept directly.

    - **ConvertTo-SecureString -AsPlainText -Force** = converts a plain text string into a `SecureString` object in memory (still visible in memory during the session, but not stored as readable clear text in variables/logs the same way).
    - **ConvertFrom-SecureString** = encrypts a SecureString and converts it to an encrypted text representation that can be safely written to disk (encrypted using DPAPI, tied to the user account and machine that encrypted it, by default).
- **Data Types & Type Casting** = PowerShell is built on .NET, so every value has an underlying .NET type, but the language is loosely/dynamically typed by default — a variable's type isn't fixed unless you explicitly declare it. Type casting is done by prefixing a value or variable with a type in square brackets.

```
[int]$Count = "5"       # casts the string "5" to an actual integer
[string]$Name = 42       # casts the integer 42 to the string "42"
[datetime]$Deployed = "2026-09-21"
$Value.GetType().Name    # returns the underlying .NET type name, e.g. Int32, String,
Boolean
```

    - **[int] / [long] / [double] / [decimal]** = whole and floating-point numeric types (long for larger integers, decimal for precise decimal math without floating-point rounding error).
    - **[string]** = text.
    - **[bool]** = $true / $false.
    - **[datetime]** = a full date/time object supporting arithmetic (`.AddDays()`, `.AddHours()`) and comparison operators directly.
    - **[array]** = a generic array type; more specific typed arrays like `[string[]]` or `[int[]]` restrict every element to that type.
    - **[hashtable]** = a dictionary/key-value collection.
    - **[pscustomobject]** = casts a hashtable into a structured, ordered custom object with real properties — the standard modern way to build clean output objects instead of returning a raw hashtable. Example: `[pscustomobject]@{ Name = "LAB-DC01"; Status = "Online" }`
    - **Declaring a variable's type explicitly** (ex. `[string]$Name`) locks that variable to only ever accept that type going forward in the script — assigning an incompatible value throws an error instead of silently converting.
- **String Methods** = because every PowerShell string is a full .NET `System.String` object, it comes with a rich set of built-in methods beyond what the shell operators (`-replace`, `-split`) provide — accessed with dot notation.

```
"  LAB-DC01  ".Trim()        # removes leading/trailing whitespace
"hello".ToUpper()                 # HELLO
"LAB-SRV03".Contains("01")         # $true
"a,b,c".Split(",")                # returns an array: a, b, c
"Hello World".Substring(0,5)      # Hello
"error: disk full".Replace("error", "warning")
```

    - **.Trim() / .TrimStart() / .TrimEnd()** = remove whitespace (or specified characters) from a string.
    - **.ToUpper() / .ToLower()** = case conversion.
    - **.Contains() / .StartsWith() / .EndsWith()** = literal (non-regex, non-wildcard) substring tests, returning $true/$false.
    - **.Split()** = splits a string into an array using a literal character delimiter (simpler than the `-split` operator's regex behavior, though `-split` is more powerful when you actually need regex).
    - **.Replace()** = a **literal** find-and-replace, unlike the `-replace` operator which always treats its pattern as regex — an important distinction when your search text contains regex special characters like `.` or `*`.
    - **.Substring(start, length)** = extracts a portion of a string by character position.
    - **.PadLeft() / .PadRight()** = pads a string to a fixed width, useful for aligning columns in plain-text report output.

    - **.IndexOf()** = returns the numeric position of the first occurrence of a substring, or -1 if not found.
- **Array & Collection Methods** = beyond simple indexing, arrays and collections expose useful built-in methods and operators for common data-shaping tasks.
    - **-join** = combines every element of an array into a single string using a specified delimiter (ex. `$Array -join ", "`) — the inverse of `-split`.
    - **.Count / .Length** = the number of elements in a collection (both are generally interchangeable on arrays).
    - **$Array | Sort-Object -Unique** = a common idiom for deduplicating and sorting a list in one step.
    - **$Array[0..2]** = the range operator inside array brackets returns a slice — elements at index 0 through 2, inclusive.
    - **[System.Collections.Generic.List[object]]::new()** = the recommended collection type when you need to frequently `.Add()` items in a loop, since (unlike a fixed-size PowerShell array with `+=`) it doesn't silently rebuild the entire collection on every append.
- **Ternary, Null-Coalescing & Pipeline Chain Operators (PowerShell 7+)** = PowerShell 7 (built on .NET Core, cross-platform) introduced several concise operators borrowed from modern languages that don't exist in Windows PowerShell 5.1.

```
$Status = $IsOnline ? "Up" : "Down"          # ternary operator
$Server = $ServerName ?? "DefaultServer"      # null-coalescing: use $ServerName
unless it's $null
$Path ??= "C:\Logs"                           # null-coalescing assignment: only
assigns if $Path is currently $null
Test-Path C:\Logs && Get-ChildItem C:\Logs     # pipeline chain: only runs the 2nd
command if the 1st succeeded
Get-Service Spooler || Write-Warning "Not found"
```

    - **? :** = the ternary conditional operator, a compact one-line alternative to a full `if/else` block for simple value assignment.
    - **??** = the null-coalescing operator; returns the left-hand value unless it's `$null`, in which case it returns the right-hand fallback value.
    - **??=** = null-coalescing assignment; only assigns a new value to the variable if it is currently `$null`.
    - **&&** = pipeline chain AND; runs the second command only if the first one succeeded (exit code 0) — conceptually identical to Bash's `&&`.
    - **||** = pipeline chain OR; runs the second command only if the first one failed — conceptually identical to Bash's `||`.
- **Advanced Functions & [CmdletBinding()]** = adding the `[CmdletBinding()]` attribute directly above a function's `param()` block transforms an ordinary function into an **advanced function** that behaves just like a real compiled cmdlet — gaining automatic support for common parameters like `-Verbose`, `-Debug`, `-ErrorAction`, and `-WhatIf`/`-Confirm` (when combined with `SupportsShouldProcess`).

```
function Remove-OldLogFile {
    [CmdletBinding(SupportsShouldProcess)]
    param(
        [Parameter(Mandatory, ValueFromPipeline)]
        [ValidateNotNullOrEmpty()]
        [string]$Path,

        [ValidateSet("Days","Weeks","Months")]
        [string]$Unit = "Days",

        [ValidateRange(1,365)]
        [int]$Age = 30
    )
    process {
        if ($PSCmdlet.ShouldProcess($Path, "Delete")) {
            Remove-Item -Path $Path -Force
        }
    }
}
```

  - **[Parameter(Mandatory)]** = forces PowerShell to prompt the user for a value if the parameter isn't supplied, instead of proceeding with a $null value.
  - **[Parameter(ValueFromPipeline)]** = allows the parameter to accept a value piped directly in from the previous command in the pipeline, exactly like native cmdlets do.
  - **[ValidateSet("a","b")]** = restricts the parameter to only a fixed list of acceptable values, rejecting (and tab-completing) anything else.
  - **[ValidateRange(min,max)]** = restricts a numeric parameter to a specific range.
  - **[ValidateNotNullOrEmpty()]** = rejects $null or empty-string values outright.
  - **SupportsShouldProcess** = enables `-WhatIf` and `-Confirm` support; you must then explicitly call `$PSCmdlet.ShouldProcess()` around the actual state-changing code for those flags to have any effect.
  - **begin / process / end blocks** = when a function accepts pipeline input, `begin` runs once before any pipeline items arrive, `process` runs once per pipeline item, and `end` runs once after all items have been processed — mirroring how compiled cmdlets are structured internally.

- **PowerShell Classes** = PowerShell 5.0+ introduced the `class` keyword for defining true .NET-backed custom object types directly in a script, supporting properties, methods, constructors, and even inheritance — used far less often than functions in day-to-day sysadmin scripting, but occasionally useful for building a reusable structured data type.

```
class Server {
    [string]$Name
    [string]$Role

    Server([string]$name, [string]$role) {
        $this.Name = $name
        $this.Role = $role
    }

    [string] GetSummary() {
        return "$($this.Name) is a $($this.Role)"
    }
}
$dc = [Server]::new("LAB-DC01", "Domain Controller")
$dc.GetSummary()
```

- **Select-String** = searches text (from a file or the pipeline) for a specified pattern — the direct PowerShell equivalent of Linux's `grep`, returning rich `MatchInfo` objects (with filename, line number, and matched text) rather than plain text lines.

```
Select-String -Path "C:\Logs\*.log" -Pattern "ERROR" -CaseSensitive
Get-Content app.log | Select-String -Pattern "timeout" -Context 2,2
```

    - **-Pattern** = the search string or regular expression (regex is the default interpretation, similar to `grep -E`).
    - **-SimpleMatch** = treats the pattern as a literal string instead of regex (the equivalent of `grep -F`).
    - **-CaseSensitive** = forces case-sensitive matching (default is case-insensitive, opposite of grep's default).
    - **-NotMatch** = inverts the match, returning only lines that do **not** match (the equivalent of `grep -v`).
    - **-Context \<before\>,\<after\>** = includes a specified number of surrounding lines of context around each match.
    - **-List** = returns only the first match per input file, rather than every match.
- **Format-Table / Format-List / Format-Wide** = control how object output is displayed in the console — note these are strictly for **display formatting only**; once an object has been formatted, it can no longer be reliably piped into another cmdlet for further processing.
    - **Format-Table (ftable/ft)** = the default display format for most collections of objects — a compact, column-based table.
    - **-AutoSize** = adjusts column widths automatically based on the actual content, rather than PowerShell's default fixed-width guess (which frequently truncates long values).
    - **-Wrap** = wraps long content within a column instead of truncating it.
    - **Format-List (fl)** = displays each object's properties as a vertical list of Name : Value pairs, one object at a time — the better choice when an object has many properties or when truncated table columns are hiding important data.
    - **Format-Wide (fw)** = displays only a single property from each object across the screen in multiple columns, similar in spirit to a bare `ls` listing filenames only.
    - **-Property *** = a very common troubleshooting habit: pipe any command through `| Format-List -Property *` to see literally every property an object has, when you're not sure what's available.
- **ConvertTo-Html** = converts pipeline objects into a formatted HTML table/document, commonly used to generate simple, shareable reports (e.g., an automated daily health-check email attachment) without needing a full reporting framework.

```
Get-Service | Where-Object Status -eq "Stopped" | ConvertTo-Html -Title "Stopped
Services Report" | Out-File C:\Reports\stopped.html
```

- **Measure-Command** = times how long a script block takes to execute, returning a detailed `TimeSpan` object — the standard way to benchmark and compare the performance of two different approaches to the same task.

```
Measure-Command { Get-ChildItem C:\Windows -Recurse }
```

- **Get-Variable / Set-Variable / Remove-Variable / Clear-Variable** = the cmdlet-based (as opposed to `$` sigil-based) way to interact with variables — mostly used when you need to work with a variable name that's stored dynamically as a string, or when auditing all currently defined variables in a session.
    - **Get-Variable** (with no arguments) = lists every variable currently defined in the session, similar in purpose to Bash's bare `set` command.
    - **Remove-Variable** = deletes a variable entirely from memory (the equivalent of Bash's `unset`).
- **$env: Provider (Environment Variables)** = PowerShell exposes environment variables through a dedicated `Env:` PSDrive, accessed with the `$env:` prefix — functionally similar to referencing environment variables directly in Bash, but requiring the explicit `env:` scope prefix since PowerShell also has its own separate variable namespace.

```
$env:COMPUTERNAME
$env:UserProfile
[System.Environment]::SetEnvironmentVariable("MyVar","Value","Machine")   # sets a
persistent system-wide variable
```

    - Simply setting `$env:VarName = "Value"` only affects the **current process** and any child processes it spawns — it does not persist after the session closes, similar to a plain (non-exported, non-persisted) Bash variable. Use `[System.Environment]::SetEnvironmentVariable()` with the `Machine` or `User` scope for a permanent change.
- **Get-TimeZone / Set-TimeZone / Get-Culture** = inspect and configure the system's time zone and regional/locale settings from PowerShell.
    - **Get-TimeZone / Set-TimeZone** = the PowerShell-native equivalent of the Linux `timedatectl set-timezone` command.
    - **Get-Culture** = shows the current locale settings (date formats, currency symbol, decimal separator) affecting how PowerShell formats output on this system.
- **New-PSDrive / Get-PSDrive** = PowerShell's provider model exposes many different data stores (filesystem paths, the registry, certificates, even a remote FTP/network share) as if they were simple drive letters — `Get-PSDrive` lists all currently mounted PSDrives, and `New-PSDrive` lets you create your own persistent mapped shortcut to a specific path.

```
New-PSDrive -Name Scripts -PSProvider FileSystem -Root "\\LAB-FS01\Scripts" -Persist
```

- **Invoke-Expression** = dynamically builds and executes a string as if it were literal PowerShell code — powerful, but generally considered risky/discouraged in production scripts (an attacker who can influence the input string can execute arbitrary code), similar in spirit to the security caution around Bash's `eval`.

```
Invoke-Expression "Get-Process $ProcName"
```

- **.NET Type Accelerators (quick reference)** = besides the basic types already covered ([int], [string], [bool], [datetime], [array], [hashtable]), PowerShell exposes many other commonly needed .NET types as short bracket accelerators instead of requiring their full namespace.
    - **[xml]** = casts a string into a navigable XML document object, letting you access elements with simple dot notation instead of manually parsing XML.
    - **[guid]** = a globally unique identifier type, commonly needed when working with AD object GUIDs or Azure resource IDs.
    - **[ipaddress]** = validates and represents an IP address as a structured object rather than a plain string.
    - **[version]** = represents a version number (major.minor.build.revision) and supports proper numeric version comparison instead of unreliable string comparison (important since string-comparing "9.10" vs "9.9" gives the wrong answer, but version-comparing them doesn't).
    - **[regex]** = gives direct access to the full .NET regular expression engine's methods (`::Match()`, `::Matches()`, `::Replace()`) beyond what the `-match`/`-replace` operators expose.
    - **[System.Net.Mail.MailMessage]**, **[System.Net.WebClient]**, and similar fully-qualified type names = used with `New-Object` (or the modern `::new()` static syntax) whenever no short accelerator exists for a needed .NET class.
- **Enums (enumerated types)** = an enum defines a fixed, named set of possible values under a single type — used constantly by native cmdlet parameters (e.g., `-ErrorAction` only accepts values from the `ErrorActionPreference` enum) and occasionally useful to define in your own scripts for clearer, self-documenting state values than plain strings.

```
enum ServerRole {
    DomainController
    FileServer
    WebServer
}
[ServerRole]$Role = "FileServer"
```

- **Windows PowerShell 5.1 vs. PowerShell 7+ — Key Differences** = a quick orientation for anyone still regularly moving between the two, since many enterprise environments (including most Windows Server management scenarios and older RSAT modules like ActiveDirectory) still run primarily on Windows PowerShell 5.1 by default.
    - **Windows PowerShell 5.1** ships built into every modern Windows OS, is Windows-only, .NET Framework-based, and is a completely separate, older codebase from PowerShell 7 — not merely an earlier version number of the same product.
    - **PowerShell 7+** is a separate, actively-developed, open-source, cross-platform product (Windows/Linux/macOS) built on modern .NET, must be installed separately (`winget install Microsoft.PowerShell` or via MSI), and runs side-by-side with Windows PowerShell 5.1 as `pwsh.exe` rather than replacing `powershell.exe`.
    - PowerShell 7 added: ternary/null-coalescing/pipeline chain operators, `ForEach-Object -Parallel`, better error messages, and cross-platform remoting over SSH in addition to WinRM.
    - Some modules — most notably `ActiveDirectory`'s older cmdlets and certain legacy WMI-dependent tools — historically ran best (or only) under Windows PowerShell 5.1, though Microsoft has been steadily closing this gap; always check current module compatibility notes rather than assuming.
- **New-Object** = creates an instance of any .NET class directly by name — a general-purpose escape hatch used whenever you need functionality from the underlying .NET Framework/Core that doesn't have a dedicated PowerShell cmdlet wrapper.

```
$WebClient = New-Object System.Net.WebClient
$StringBuilder = New-Object System.Text.StringBuilder
```

    - The more modern static syntax `[System.Text.StringBuilder]::new()` is now generally preferred over `New-Object` for performance and readability, but you'll see `New-Object` constantly in existing production scripts.
- **$PSVersionTable** = an automatic hashtable-like variable showing the exact PowerShell version, edition (Desktop vs Core), and CLR/.NET version of the current session — the first thing to check when a script behaves differently on two machines. (ex. `$PSVersionTable.PSVersion`)
- **.. (Range Operator)** = generates a sequential array of integers (or characters) between two endpoints, ascending or descending. (ex. `1..5` produces `1,2,3,4,5`; `5..1` counts down)
- **Compound Assignment & Increment/Decrement Operators** = shorthand operators for updating a variable's value in place, identical in concept to Bash/C-style languages.
    - **+= -= *= /= %=** = perform the operation on the right against the variable's current value and store the result back into it. (ex. `$Count += 1` is shorthand for `$Count = $Count + 1`)
    - **++ / --** = increment or decrement a numeric variable by exactly 1. (ex. `$i++`)
- **Containment Operators (-in / -notin / -contains / -notcontains)** = test whether a value exists inside a collection, which reads more naturally than a `Where-Object` filter for a simple membership check.
    - **-in / -notin** = the value is on the left, the collection is on the right. (ex. `if ("web01" -in $ServerList)`)
    - **-contains / -notcontains** = the collection is on the left, the value is on the right — the mirror image of `-in`/`-notin`. (ex. `if ($ServerList -contains "web01")`)
- **Type Operators (-is / -as)** = check or convert an object's .NET type at runtime.
    - **-is** = returns `$true`/`$false` for whether an object is a given type. (ex. `$Value -is [int]`)

    - **-as** = attempts to convert an object to the given type, returning `$null` instead of throwing an error if the conversion fails (unlike a hard `[int]$Value` cast).
- **return** = immediately exits a function or script and optionally sends a value back to the caller. Unlike many languages, PowerShell functions also implicitly return anything not otherwise captured or suppressed (like unassigned pipeline output), so `return` is really about controlling *flow*, not the only way to produce output.
- **$args** = an automatic array variable inside a function or script that holds any arguments passed without matching a named parameter — PowerShell's loose equivalent of Bash's `$@`, though named parameters are strongly preferred in real scripts.
- **begin / process / end Blocks** = the three named script blocks a function can declare (alongside `[CmdletBinding()]`) to properly support pipeline input.
    - **begin** = runs exactly once, before any pipeline input arrives — used for setup like opening a connection or initializing a counter.
    - **process** = runs once for *each* object received from the pipeline — this is where the per-item work happens.
    - **end** = runs exactly once, after all pipeline input has been processed — used for cleanup or printing a summary.
- **Parameter Validation Attributes** = decorate a function's `param()` block to reject bad input before the function body ever runs, producing a much clearer error than a generic crash mid-script.
    - **[Parameter(Mandatory=$true)]** = forces PowerShell to prompt for the parameter (or fail) if it isn't supplied.
    - **[ValidateSet("A","B")]** = restricts the parameter to one of a fixed list of allowed values, with tab-completion for free.
    - **[ValidateRange(1,100)]** = restricts a numeric parameter to a min/max range.
    - **[ValidateNotNullOrEmpty()]** = rejects `$null` or an empty string/collection immediately.

Real-World Scripting Cookbook

These are complete, ready-to-adapt scripts combining the cmdlets and syntax covered throughout this guide into the kind of everyday automation a sysadmin actually writes — bulk AD operations, health-check reports, cleanup jobs, and cross-module scripts spanning AD, Graph, Azure, and Exchange Online together. Treat these as starting templates, not drop-in production code — always test with -WhatIf and against a non-production scope first.

- **Bulk-create AD users from a CSV file** = one of the most common onboarding automation tasks: reading a prepared CSV of new hires and creating each AD account in a single pass instead of manually clicking through ADUC for every new employee.

```
$NewHires = Import-Csv -Path "C:\Onboarding\NewHires.csv"

foreach ($Hire in $NewHires) {
    $Params = @{
        Name              = "$($Hire.FirstName) $($Hire.LastName)"
        GivenName         = $Hire.FirstName
        Surname           = $Hire.LastName
        SamAccountName    = $Hire.Username
        UserPrincipalName = "$($Hire.Username)@contoso.com"
        Path              = "OU=Users,OU=$($Hire.Department),DC=contoso,DC=com"
        AccountPassword   = (ConvertTo-SecureString $Hire.TempPassword -AsPlainText
-Force)
        Enabled           = $true
        ChangePasswordAtLogon = $true
    }
    New-ADUser @Params
    Add-ADGroupMember -Identity $Hire.Department -Members $Hire.Username
}
```

- **Find and report stale/inactive computer accounts** = finds computer accounts that haven't authenticated to the domain in over 90 days — a standard periodic cleanup report to identify decommissioned or forgotten devices that should be disabled or removed from AD.

```
$CutoffDate = (Get-Date).AddDays(-90)
Get-ADComputer -Filter { LastLogonTimestamp -lt $CutoffDate } -Properties
LastLogonDate, OperatingSystem |
    Select-Object Name, OperatingSystem, LastLogonDate |
    Sort-Object LastLogonDate |
    Export-Csv -Path "C:\Reports\StaleComputers.csv" -NoTypeInformation
```

- **Disk space health-check report across multiple servers** = queries free disk space across a list of servers and flags anything below a critical threshold — the kind of quick scheduled report that catches a filling drive before it becomes an outage.

```
$Servers = Get-Content "C:\Scripts\ServerList.txt"
$Report = foreach ($Server in $Servers) {
    Get-CimInstance -ClassName Win32_LogicalDisk -ComputerName $Server -Filter
"DriveType=3" |
    ForEach-Object {
        [pscustomobject]@{
            Server     = $Server
            Drive      = $_.DeviceID
            FreeGB     = [math]::Round($_.FreeSpace / 1GB, 1)
            SizeGB     = [math]::Round($_.Size / 1GB, 1)
            PercentFree = [math]::Round(($_.FreeSpace / $_.Size) * 100, 1)
        }
    }
}
$Report | Where-Object PercentFree -lt 15 | Sort-Object PercentFree
```

- **Email alert when a critical Windows service stops** = a self-contained monitoring script (typically run as a scheduled task every few minutes) that checks a list of critical services and sends an email alert if any are found stopped — a lightweight alternative to a full monitoring platform for a handful of key services.

```
$CriticalServices = "Spooler","W32Time","DNS","Netlogon"
$Down = Get-Service -Name $CriticalServices | Where-Object Status -ne "Running"

if ($Down) {
    $Body = ($Down | Select-Object Name, Status | Out-String)
    Send-MailMessage -From "alerts@contoso.com" -To "itops@contoso.com" `
        -Subject "ALERT: Critical service(s) stopped on $env:COMPUTERNAME" `
        -Body $Body -SmtpServer "smtp.contoso.com"
}
```

    - **Send-MailMessage** is officially **deprecated** by Microsoft (though still functional in Windows PowerShell 5.1) due to lacking modern authentication support; new scripts should send mail via `Invoke-RestMethod` against the Microsoft Graph `sendMail` endpoint, or via a dedicated SMTP relay/connector that doesn't require this cmdlet's legacy auth model.

- **Certificate expiration report across all servers** = scans the local machine certificate store on a list of servers and reports any certificate expiring within the next 30 days — the scripted equivalent of the Linux workflow checking `openssl x509 -enddate` against a threshold, run centrally against many machines at once via remoting.

```
$Servers = Get-Content "C:\Scripts\ServerList.txt"
Invoke-Command -ComputerName $Servers -ScriptBlock {
    Get-ChildItem -Path Cert:\LocalMachine\My |
    Where-Object { $_.NotAfter -lt (Get-Date).AddDays(30) } |
    Select-Object Subject, NotAfter, Thumbprint
} | Select-Object PSComputerName, Subject, NotAfter, Thumbprint |
    Export-Csv "C:\Reports\CertExpiration.csv" -NoTypeInformation
```

- **Bulk-disable AD accounts and move them to a Disabled OU (offboarding)** = a standard offboarding script: disables an account, strips its group memberships (recording them first for audit purposes), and moves it to a dedicated 'Disabled Users' OU where a separate cleanup job later deletes anything sitting there for 90+ days.

```
$User = "asample"
$Groups = Get-ADPrincipalGroupMembership -Identity $User | Where-Object Name -ne
"Domain Users"
$Groups.Name | Out-File "C:\Offboarding\$User-Groups.txt"
$Groups | ForEach-Object { Remove-ADGroupMember -Identity $_ -Members $User
-Confirm:$false }

Disable-ADAccount -Identity $User
Set-ADUser -Identity $User -Description "Disabled $(Get-Date -Format yyyy-MM-dd) -
offboarded"
Move-ADObject -Identity (Get-ADUser $User).DistinguishedName -TargetPath "OU=Disabled
Users,DC=contoso,DC=com"
```

- **Find all users with passwords about to expire and notify them** = identifies AD users whose password will expire within the next 7 days and emails each of them a reminder — reduces helpdesk lockout tickets by giving users advance warning instead of relying solely on the built-in Windows logon warning.

```
$MaxPasswordAge = (Get-ADDefaultDomainPasswordPolicy).MaxPasswordAge.Days
$Users = Get-ADUser -Filter { Enabled -eq $true -and PasswordNeverExpires -eq $false }
-Properties PasswordLastSet, EmailAddress

foreach ($U in $Users) {
    $DaysLeft = $MaxPasswordAge - ((Get-Date) - $U.PasswordLastSet).Days
    if ($DaysLeft -le 7 -and $DaysLeft -gt 0) {
        Send-MailMessage -From "itops@contoso.com" -To $U.EmailAddress `
            -Subject "Your password expires in $DaysLeft day(s)" `
            -Body "Please change your password before it expires to avoid being locked
out." `
            -SmtpServer "smtp.contoso.com"
    }
}
```

- **Automated log cleanup: delete files older than N days** = a common scheduled-task script for keeping a log directory from growing unbounded — the PowerShell equivalent of the `logrotate`/`find -mtime +N -delete` pattern used constantly on Linux.

```
$LogPath = "C:\Logs"
$RetentionDays = 30

Get-ChildItem -Path $LogPath -Filter "*.log" -Recurse |
    Where-Object { $_.LastWriteTime -lt (Get-Date).AddDays(-$RetentionDays) } |
    Remove-Item -Force -WhatIf
```

    - Remove the trailing `-WhatIf` only after confirming the exact set of files it would delete looks correct — this is precisely the kind of destructive bulk operation where testing first matters most.

- **Query Microsoft Graph for all guest accounts in the tenant** = a periodic external-access security audit: lists every guest (B2B) account in Entra ID along with when they were created and their current sign-in activity, to catch stale guest access that should be reviewed or revoked.

```
Connect-MgGraph -Scopes "User.Read.All"
Get-MgUser -Filter "userType eq 'Guest'" -All -Property
DisplayName,Mail,CreatedDateTime,SignInActivity |
    Select-Object DisplayName, Mail, CreatedDateTime,
@{N='LastSignIn';E={$_.SignInActivity.LastSignInDateTime}} |
    Export-Csv "C:\Reports\GuestAccounts.csv" -NoTypeInformation
```

- **Bulk-add a distribution list or Microsoft 365 Group from a CSV of members** = creates a new distribution group and populates it from a CSV file of email addresses — useful when standing up a new site or department alias in one pass instead of adding members one at a time.

```
Connect-ExchangeOnline
$Members = Import-Csv "C:\Groups\NewDLMembers.csv"
New-DistributionGroup -Name "Site-A-Staff" -PrimarySmtpAddress "site-a@contoso.com"
foreach ($M in $Members) {
    Add-DistributionGroupMember -Identity "Site-A-Staff" -Member $M.EmailAddress
}
```

- **Check replication health across all domain controllers** = a scheduled AD health check that surfaces any domain controller with a replication failure, without needing to manually run `repadmin` against every DC by hand.

```
$DCs = Get-ADDomainController -Filter *
$Report = foreach ($DC in $DCs) {
    Get-ADReplicationPartnerMetadata -Target $DC.HostName |
    Select-Object Server, Partner, LastReplicationSuccess, LastReplicationResult,
ConsecutiveReplicationFailures
}
$Report | Where-Object ConsecutiveReplicationFailures -gt 0
```

- **Deploy a scheduled task to a fleet of servers via remoting** = creates the same scheduled task across a whole list of remote servers in one pass — much faster and more consistent than logging into each server individually to configure Task Scheduler by hand.

```
$Servers = Get-Content "C:\Scripts\ServerList.txt"
$Action  = New-ScheduledTaskAction -Execute "powershell.exe" -Argument "-File
C:\Scripts\CleanTemp.ps1"
$Trigger = New-ScheduledTaskTrigger -Daily -At "3:00AM"

Invoke-Command -ComputerName $Servers -ScriptBlock {
    param($Action, $Trigger)
    Register-ScheduledTask -TaskName "Nightly Temp Cleanup" -Action $Action -Trigger
$Trigger -User "SYSTEM" -Force
} -ArgumentList $Action, $Trigger
```

- **Test-and-repair pattern for a fleet of servers** = a common resilient automation pattern: attempt something, catch any failures per-server without stopping the whole batch, and report a clean summary of successes and failures at the end.

```
$Servers = Get-Content "C:\Scripts\ServerList.txt"
$Results = foreach ($Server in $Servers) {
    try {
        $null = Test-WSMan -ComputerName $Server -ErrorAction Stop
        [pscustomobject]@{ Server = $Server; Status = "Reachable" }
    } catch {
        [pscustomobject]@{ Server = $Server; Status = "UNREACHABLE:
$($_.Exception.Message)" }
    }
}
$Results | Format-Table -AutoSize
```

- **Application-only Graph authentication for a scheduled/unattended script** = the standard pattern for a script that must run unattended (via Task Scheduler or Azure Automation) against Microsoft Graph, using a certificate-backed App Registration instead of an interactive user sign-in that would require MFA every time.

```
$AppId       = "11111111-2222-3333-4444-555555555555"
$TenantId    = "66666666-7777-8888-9999-aaaaaaaaaaaa"
$Thumbprint  = "ABCDEF1234567890ABCDEF1234567890ABCDEF12"

Connect-MgGraph -ClientId $AppId -TenantId $TenantId -CertificateThumbprint
$Thumbprint -NoWelcome
Get-MgUser -Top 5 | Select-Object DisplayName, UserPrincipalName
Disconnect-MgGraph
```

    - The App Registration must be pre-granted the specific Graph **Application** permissions it needs (e.g., `User.Read.All`) via admin consent — application permissions are a fixed, tenant-wide grant, unlike delegated scopes which are bound to whatever the signed-in user is already allowed to do.

- **Weekly report of newly created Azure resources (change tracking)** = pulls the Azure Activity Log for the past 7 days and reports on resource creation events, useful as a lightweight, scripted change-tracking mechanism for a subscription without needing a dedicated CMDB integration.

```
Connect-AzAccount -Identity
$Log = Get-AzActivityLog -StartTime (Get-Date).AddDays(-7) |
    Where-Object { $_.OperationName.Value -like "*write" -and $_.Status.Value -eq
"Succeeded" }
$Log | Select-Object EventTimestamp, Caller, ResourceId | Export-Csv
"C:\Reports\AzureChanges.csv" -NoTypeInformation
```

- **Bulk mailbox permission audit report** = generates a full report of every non-default Full Access delegate grant across all mailboxes in the tenant — a standard periodic access review to confirm delegate access hasn't grown beyond what's actually approved.

```
Connect-ExchangeOnline
$Report = Get-EXOMailbox -ResultSize Unlimited | ForEach-Object {
    Get-EXOMailboxPermission -Identity $_.UserPrincipalName |
    Where-Object { $_.User -ne "NT AUTHORITY\SELF" -and -not $_.IsInherited }
}
$Report | Select-Object Identity, User, AccessRights | Export-Csv
"C:\Reports\MailboxPermissions.csv" -NoTypeInformation
```

- **Find which process is holding a specific TCP port** = cross-references active network connections with running processes to answer 'what is actually listening on this port' — the PowerShell-native equivalent of piping `netstat -ano` through `findstr` and then manually looking up the PID in Task Manager.

```
$Port = 443
Get-NetTCPConnection -LocalPort $Port -State Listen |
    ForEach-Object {
        [pscustomobject]@{
            Port    = $_.LocalPort
            PID     = $_.OwningProcess
            Process = (Get-Process -Id $_.OwningProcess).ProcessName
        }
    }
```

- **Live-tail a remote log file over PowerShell Remoting** = streams new lines from a log file on a remote server directly to your local console in real time — combining `Get-Content -Wait` with `Invoke-Command`, giving you the equivalent of `ssh server tail -f logfile` on Linux.

```
Invoke-Command -ComputerName LAB-SRV01 -ScriptBlock { Get-Content
"C:\inetpub\logs\LogFiles\app.log" -Wait -Tail 20 }
```

- **Reconcile installed software across a fleet against an approved list** = queries installed software from the registry (faster and more reliable than the notoriously slow `Win32_Product` WMI class) across multiple servers and flags anything not on an approved software baseline — useful for both security auditing and license compliance.

```
$Approved = Get-Content "C:\Scripts\ApprovedSoftware.txt"
$Servers  = Get-Content "C:\Scripts\ServerList.txt"

Invoke-Command -ComputerName $Servers -ScriptBlock {
    Get-ItemProperty "HKLM:\Software\Microsoft\Windows\CurrentVersion\Uninstall\*" |
    Select-Object DisplayName, DisplayVersion | Where-Object DisplayName
} | Where-Object { $_.DisplayName -notin $using:Approved } |
    Select-Object PSComputerName, DisplayName, DisplayVersion
```

    - Querying the registry uninstall keys directly (as shown here) is dramatically faster than `Get-CimInstance Win32_Product`, which triggers a costly consistency-check/repair scan of every installed MSI package as a side effect of simply being queried.

- **Automated password reset with a random, secure temporary password** = generates a cryptographically random temporary password (rather than a predictable pattern) and resets an AD account's password, forcing a change at next logon — a safer helpdesk-automation building block than a hardcoded/predictable default password.

```
function New-RandomPassword {
    param([int]$Length = 16)
    $Chars = 'ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnpqrstuvwxyz23456789!@#$%'
    -join (1..$Length | ForEach-Object { $Chars[(Get-Random -Maximum
$Chars.Length)] })
}

$NewPassword = New-RandomPassword
Set-ADAccountPassword -Identity asample -Reset -NewPassword (ConvertTo-SecureString
$NewPassword -AsPlainText -Force)
Set-ADUser -Identity asample -ChangePasswordAtLogon $true
Write-Output "Temporary password for asample: $NewPassword"
```

- **Bulk-export all Conditional Access policies for backup/change tracking** = exports every Conditional Access policy in the tenant to individual JSON files — a lightweight way to maintain a version-controlled backup of CA configuration before making changes, since CA policies have no built-in 'undo' if a change locks out administrators.

```
Connect-MgGraph -Scopes "Policy.Read.All"
$Policies = Get-MgIdentityConditionalAccessPolicy -All
foreach ($Policy in $Policies) {
    $Policy | ConvertTo-Json -Depth 10 | Out-File "C:\CABackups\$($Policy.DisplayName
-replace '[\\/:*?"<>|]','_').json"
}
```

- **Find all locked-out accounts and who/where they were locked out from** = identifies currently locked-out accounts and cross-references the Security event log on domain controllers to find the specific source computer that triggered the lockout — the standard first two steps for any 'why does my account keep locking out' ticket, which is almost always caused by a stale cached credential somewhere (a mapped drive, scheduled task, or mobile device).

```
Search-ADAccount -LockedOut | Select-Object Name, SamAccountName

Get-WinEvent -ComputerName LAB-DC01 -FilterHashtable @{LogName='Security'; Id=4740}
-MaxEvents 20 |
    Select-Object TimeCreated, @{N='LockedUser';E={$_.Properties[0].Value}},
@{N='SourceComputer';E={$_.Properties[1].Value}}
```

- **Generate a printer fleet inventory report from asset-management data** = a pattern for consolidating printer fleet data queried directly from each device or a management server into one clean report — useful when reconciling a physical audit against asset inventory records across many sites.

```
$PrintServers = "LAB-PRINT01","LAB-PRINT02"
$Report = foreach ($PS in $PrintServers) {
    Get-Printer -ComputerName $PS | ForEach-Object {
        [pscustomobject]@{
            Server       = $PS
            PrinterName  = $_.Name
            DriverName   = $_.DriverName
            PortName     = $_.PortName
            Shared       = $_.Shared
        }
    }
}
$Report | Export-Csv "C:\Reports\PrinterFleet.csv" -NoTypeInformation
```

- **Bulk-set a DKIM/SPF/DMARC verification report across accepted domains** = checks DNS TXT/CNAME records for every accepted domain in the tenant against expected DKIM/DMARC configuration — combining Exchange Online's own domain list with a live DNS lookup, useful for periodically confirming email authentication records haven't drifted or been accidentally removed.

```
Connect-ExchangeOnline
$Domains = Get-AcceptedDomain
foreach ($Domain in $Domains) {
    $Dkim = Get-DkimSigningConfig -Identity $Domain.DomainName -ErrorAction
SilentlyContinue
    $Dmarc = Resolve-DnsName -Name "_dmarc.$($Domain.DomainName)" -Type TXT
-ErrorAction SilentlyContinue
    [pscustomobject]@{
        Domain       = $Domain.DomainName
        DkimEnabled  = $Dkim.Enabled
        DmarcRecord  = ($Dmarc.Strings -join '; ')
    }
}
```

- **Automated VM snapshot before a risky change (Hyper-V)** = a safety-net pattern for wrapping a risky maintenance script with an automatic pre-change checkpoint, so a bad update or config change can be rolled back in seconds instead of triggering a full restore from backup.

```
$VMName = "LAB-DC01"
Checkpoint-VM -Name $VMName -SnapshotName "Pre-EntraConnect-Migration-$(Get-Date
-Format yyyyMMdd-HHmm)"

# ... perform the risky change here ...

# If something goes wrong:
# Get-VMSnapshot -VMName $VMName | Sort-Object CreationTime -Descending |
Select-Object -First 1 | Restore-VMSnapshot -Confirm:$false
```

- **Audit local Administrators group membership across a server fleet** = a security baseline check that reports exactly who has local admin rights on every server in a list — critical for catching accounts that were granted temporary elevated access and never removed.

```
$Servers = Get-Content "C:\Scripts\ServerList.txt"
Invoke-Command -ComputerName $Servers -ScriptBlock {
    Get-LocalGroupMember -Group "Administrators" | Select-Object Name, ObjectClass
} | Select-Object PSComputerName, Name, ObjectClass |
    Export-Csv "C:\Reports\LocalAdmins.csv" -NoTypeInformation
```

- **Full server pre-maintenance health snapshot (multi-source report)** = a comprehensive pre-change snapshot combining several data sources into a single report — the kind of baseline you'd capture immediately before a risky maintenance window (like a migration or major patch) so you have a clear 'before' state to compare against if something goes wrong afterward.

```
$Server = "LAB-DC01"
$Snapshot = [ordered]@{
    Timestamp       = Get-Date
    Uptime          = (Get-Date) - (Get-CimInstance Win32_OperatingSystem
-ComputerName $Server).LastBootUpTime
    Services        = Get-Service -ComputerName $Server | Where-Object StartType -eq
"Automatic" | Where-Object Status -ne "Running"
    DiskSpace       = Get-CimInstance Win32_LogicalDisk -ComputerName $Server -Filter
"DriveType=3" |
                        Select-Object DeviceID,
@{N='FreeGB';E={[math]::Round($_.FreeSpace/1GB,1)}}
    RecentEvents    = Get-WinEvent -ComputerName $Server -FilterHashtable
@{LogName='System'; Level=1,2; StartTime=(Get-Date).AddHours(-24)} -MaxEvents 25
    Replication     = Get-ADReplicationPartnerMetadata -Target $Server
}
$Snapshot | ConvertTo-Json -Depth 5 | Out-File
"C:\PreChangeSnapshots\$Server-$(Get-Date -Format yyyyMMdd-HHmm).json"
```

- **Notify a Teams channel from a PowerShell script (Incoming Webhook)** = posts a message to a Microsoft Teams channel directly from a script using a channel's configured Incoming Webhook connector — a lightweight way to get automation status updates (backup completion, health check failures) into a team's chat without needing a full bot registration.

```
$WebhookUrl = "https://contoso.webhook.office.com/webhookb2/..."
$Payload = @{ text = "Nightly backup job completed successfully on $env:COMPUTERNAME
at $(Get-Date)" } | ConvertTo-Json
Invoke-RestMethod -Uri $WebhookUrl -Method Post -Body $Payload -ContentType
"application/json"
```

- **Rotate a gMSA-backed scheduled task credential-free service check** = confirms a Group Managed Service Account is actually usable from a given server before relying on it for a scheduled task — catching the common failure mode where a gMSA was created and its password-retrieval group configured correctly, but `Install-ADServiceAccount` was never run on the specific server that needs to use it.

```
$Servers = "LAB-APP01","LAB-APP02","LAB-APP03"
Invoke-Command -ComputerName $Servers -ScriptBlock {
    Test-ADServiceAccount -Identity "gMSA-App01"
} | ForEach-Object { [pscustomobject]@{ Server = $_.PSComputerName; gMSAUsable =
$_ } }
```

- **Combine on-prem AD and Entra ID data into a single identity report** = cross-references an on-premises AD user with their corresponding Entra ID object side by side — useful for troubleshooting sync issues (Entra Connect/Cloud Sync) or auditing licensing against actual on-prem account status in one unified view.

```
Import-Module ActiveDirectory
Connect-MgGraph -Scopes "User.Read.All"

$ADUser  = Get-ADUser -Identity asample -Properties Enabled, LastLogonDate
$MgUser  = Get-MgUser -UserId "asample@contoso.com" -Property AccountEnabled,
SignInActivity

[pscustomobject]@{
    SamAccountName   = $ADUser.SamAccountName
    ADEnabled        = $ADUser.Enabled
    ADLastLogon      = $ADUser.LastLogonDate
    EntraEnabled     = $MgUser.AccountEnabled
    EntraLastSignIn  = $MgUser.SignInActivity.LastSignInDateTime
}
```

Cmdlets

These are the native, built-in PowerShell cmdlets you'll use constantly regardless of which specialized module you're working in — file system navigation, process and service management, remoting, formatting, and system/hardware inspection via CIM/WMI. Think of this section as PowerShell's answer to the Linux 'Commonly Used Commands' section.

- **Get-Help / Get-Command / Get-Member** = already covered in detail in the Scripting Fundamentals section above — listed here again only as a reminder that these three are the essential discovery trio for exploring any unfamiliar cmdlet or object.
- **Get-ChildItem** = lists the contents of a directory (or, more generally, any PowerShell 'provider' path such as the registry or a certificate store) — the direct equivalent of Linux's `ls`. Alias: `dir`, `ls`, `gci`.
  - **-Recurse** = lists contents of all subdirectories as well, not just the top-level directory.
  - **-Filter** = filters results using a provider-native wildcard filter (faster, since filtering happens before objects come back), ex. `-Filter *.log`
  - **-Include / -Exclude** = filters results after retrieval; more flexible than `-Filter` (supports multiple patterns) but slower on very large directories.
  - **-Force** = shows hidden and system files, which are excluded by default.
  - **-Directory / -File** = restricts results to only directories or only files.
  - **-Attributes** = filters by specific file attribute flags (ReadOnly, Hidden, System, Archive, etc.).
- **Get-Content / Set-Content / Add-Content** = read and write the contents of a text file — the PowerShell equivalents of `cat`, and `echo >` / `echo >>` respectively.
  - **Get-Content \<path\>** = reads a file's contents into the pipeline, one line per object by default. Alias: `gc`, `cat`, `type`.
  - **-Tail \<n\>** = returns only the last N lines of a file — the direct equivalent of Linux's `tail -n`.
  - **-Wait** = keeps the file open and streams new lines as they're written, exactly like `tail -f` on Linux — invaluable for watching a live log file.
  - **-Raw** = returns the entire file as a single multi-line string instead of an array of line-strings (important when you need to preserve exact formatting or run a multi-line regex against the whole file).
  - **Set-Content** = overwrites a file's contents (equivalent to Bash's `>` redirection). **Add-Content** = appends to a file's contents (equivalent to Bash's `>>` redirection).
- **New-Item / Remove-Item / Copy-Item / Move-Item / Rename-Item** = the core file/directory manipulation cmdlets — the PowerShell equivalents of `mkdir`/`touch`, `rm`, `cp`, `mv`, and a rename operation respectively.
  - **New-Item -Path \<path\> -ItemType Directory|File** = creates a new folder or empty file.
  - **Remove-Item -Recurse -Force** = deletes a file or directory (and everything inside it recursively, forcibly without prompting).
  - **Copy-Item -Recurse** = copies files/folders, including subdirectory contents when `-Recurse` is specified.
  - **Move-Item** = moves (or effectively renames, if the destination is in the same folder with a new name) a file or folder.
  - **Rename-Item -NewName \<name\>** = renames a file or folder without needing to specify a full new path.
  - **-WhatIf** = available on nearly every state-changing cmdlet in PowerShell; shows what *would* happen without actually performing the action — an extremely valuable safety habit before running any destructive command against production.
  - **-Confirm** = forces an interactive Y/N confirmation prompt before the action executes.
- **Test-Path** = checks whether a file, folder, or registry key path exists, returning a simple $true/$false — used constantly at the start of scripts to validate assumptions before proceeding.

    - **-PathType Leaf|Container** = restricts the check to specifically a file (Leaf) or specifically a directory (Container).
- **Get-Item / Set-Item** = retrieves (or sets the value of) a single item at a specific provider path — this generic pair works not just on files, but also on registry keys, environment variables, and PowerShell drives, since it operates through the PowerShell provider model rather than being filesystem-specific.
    - Example against the registry: `Get-Item "HKLM:\SOFTWARE\Microsoft\Windows NT\CurrentVersion"`
    - Example against an environment variable: `Get-Item Env:\PATH`
- **Get-ItemProperty / Set-ItemProperty** = retrieves or sets a specific **property** (value) *of* an item — most commonly used against the registry to read or write a specific registry value name inside a key (as opposed to `Get-Item`, which retrieves the key itself).

```
Get-ItemProperty -Path "HKLM:\SOFTWARE\MyApp" -Name "Version"
Set-ItemProperty -Path "HKLM:\SOFTWARE\MyApp" -Name "Version" -Value "2.0"
```

- **Get-Process** = lists currently running processes on the local machine, with CPU, memory, and handle information — the PowerShell equivalent of `ps`/`tasklist`. Alias: `gps`, `ps`.
    - **-Name \<name\>** = filters to processes matching a specific name (wildcards supported).
    - **-Id \<PID\>** = filters to a specific Process ID.
    - **-ComputerName** = (in Windows PowerShell 5.1) queries a remote computer's process list directly; in PowerShell 7+, use `Invoke-Command` instead since this parameter was removed.
    - **-IncludeUserName** = includes the owning username for each process (requires elevated privileges).
- **Stop-Process** = terminates a running process — the PowerShell equivalent of `kill`/`taskkill`.
    - **-Id \<PID\>** or **-Name \<name\>** = specifies the target process.
    - **-Force** = forcibly terminates the process without prompting for confirmation.
- **Start-Process** = launches a new process, with fine control over window state, credentials, and whether to wait for it to exit — the PowerShell equivalent of running a program directly, but with much richer control (e.g., launching elevated, redirecting output, or waiting for completion in a script).
    - **-Verb RunAs** = launches the process elevated (triggering a UAC prompt), the scripted equivalent of right-click 'Run as administrator'.
    - **-Wait** = pauses script execution until the launched process exits, instead of continuing immediately.
    - **-NoNewWindow** = runs the process without opening a new console window.
    - **-ArgumentList** = passes command-line arguments to the launched executable.
    - **-RedirectStandardOutput / -RedirectStandardError** = captures the process's console output to files instead of displaying it.
- **Get-Service / Start-Service / Stop-Service / Restart-Service / Set-Service** = manage Windows services — the PowerShell equivalent of the `sc` command and `systemctl` on Linux for service lifecycle management.
    - **Get-Service -Name \<name\>** = shows the status (Running/Stopped) of a service. Wildcards are supported in the name.
    - **Start-Service / Stop-Service / Restart-Service** = control the running state of a service, directly analogous to `systemctl start/stop/restart`.
    - **Set-Service -StartupType Automatic|Manual|Disabled** = configures whether a service starts automatically at boot, must be started manually, or is disabled entirely (the equivalent of `systemctl enable`/`disable`).
    - **-Force** = with `Stop-Service`, also stops any dependent services that rely on the target service.
    - **Get-Service -ComputerName \<name\>** = (Windows PowerShell 5.1 only) queries services on a remote computer directly; in PowerShell 7+, wrap the call in `Invoke-Command` instead.

- **Get-CimInstance** = the modern cmdlet for querying WMI (Windows Management Instrumentation) classes — hardware inventory, OS details, installed software, disk info, and hundreds of other system data points. This is the direct, non-deprecated replacement for both `Get-WmiObject` and the legacy `wmic.exe` command-line tool.

```
Get-CimInstance -ClassName Win32_OperatingSystem | Select-Object Caption, Version,
LastBootUpTime
Get-CimInstance -ClassName Win32_LogicalDisk | Select-Object DeviceID,
@{N='FreeGB';E={[math]::Round($_.FreeSpace/1GB,2)}}
```

  - **-ClassName** = the WMI class to query (common ones: `Win32_OperatingSystem`, `Win32_ComputerSystem`, `Win32_LogicalDisk`, `Win32_BIOS`, `Win32_Processor`, `Win32_PhysicalMemory`, `Win32_NetworkAdapterConfiguration`, `Win32_Product`).
  - **-Filter** = applies a WQL (WMI Query Language) filter string server-side, more efficient than filtering after retrieval with `Where-Object`.
  - **-ComputerName** = queries one or more remote computers, using the modern WSMan/WinRM protocol under the hood instead of legacy DCOM (which `Get-WmiObject` used, and which is often blocked by modern firewalls).
  - **Invoke-CimMethod** = calls a method on a CIM class or instance (ex. calling the `Create` method on `Win32_Process` to start a remote process, or `Terminate` to kill one).
- **Get-WmiObject** = the legacy predecessor to `Get-CimInstance`, using the older DCOM protocol for remote queries. **Deprecated and removed entirely in PowerShell 7+** (Windows PowerShell 5.1 only) — new scripts should always use `Get-CimInstance` instead, but you'll still encounter this constantly in older production scripts.
- **Get-EventLog** = the legacy cmdlet for reading classic Windows Event Logs (Application, System, Security). **Deprecated** in favor of `Get-WinEvent`, and not available at all in PowerShell 7+ on non-Windows platforms, but still common in older monitoring scripts.
  - **-LogName \<name\>** = specifies which log to read (Application, System, Security).
  - **-EntryType Error|Warning|Information** = filters by severity type.
  - **-Newest \<n\>** = returns only the N most recent entries.
- **Get-WinEvent** = the modern, fully-featured cmdlet for querying Windows Event Logs, including the newer structured logs under 'Applications and Services Logs' (like the ones covering Directory Service replication or DNS Server events) that `Get-EventLog` cannot access at all.
  - **-LogName \<name\>** = queries a specific classic or modern log by name (ex. `'"Directory Service"`, `'"Microsoft-Windows-DNSServer/Analytical"`).
  - **-FilterHashtable** = the high-performance way to filter events server-side before they're returned, using a hashtable of criteria (LogName, ProviderName, Id, Level, StartTime, EndTime). This is dramatically faster than pulling everything and piping to `Where-Object` on a busy server.
  - **-MaxEvents \<n\>** = limits the number of events returned.
  - **-FilterXPath** = an alternative, even more precise filtering mechanism using XPath query syntax against the event XML.
  - Example: `Get-WinEvent -FilterHashtable @{LogName='Security'; Id=4625; StartTime=(Get-Date).AddDays(-1)}` — pulls failed logon events (Event ID 4625) from the last 24 hours.
- **Test-Connection** = the PowerShell equivalent of `ping`, returning structured objects instead of plain text — allowing you to programmatically check reachability, latency, and packet loss in a script rather than parsing text output.
  - **-Count \<n\>** = number of pings to send (default is 4, matching Windows `ping`'s default).
  - **-Quiet** = returns a simple $true/$false instead of detailed ping results — extremely common inside `if` statements and `while` loop conditions.
  - **-ComputerName** = accepts an array of hostnames, letting you test multiple targets in a single command.

- **Test-NetConnection** = a more powerful network diagnostic cmdlet that combines ping-style reachability testing with specific TCP port testing and basic traceroute functionality all in one — the PowerShell equivalent of combining `ping`, `nc -z`, and `tracert`.
    - **-ComputerName \<host\> -Port \<port\>** = tests whether a specific TCP port is reachable on a remote host (the most common modern replacement for the old habit of using telnet just to test port connectivity).
    - **-TraceRoute** = includes a hop-by-hop traceroute to the destination in the output.
    - **-InformationLevel Detailed** = returns full diagnostic detail instead of the abbreviated default summary.
- **Resolve-DnsName** = queries DNS for a hostname or record — the PowerShell equivalent of `nslookup`/`dig`, returning structured record objects instead of plain text.
    - **-Type \<recordtype\>** = specifies the DNS record type to query (A, AAAA, MX, TXT, NS, CNAME, SOA, ANY).
    - **-Server \<dnsserver\>** = queries a specific DNS server instead of the system's configured default.
- **Get-NetIPAddress / Get-NetIPConfiguration / Get-NetAdapter** = modern PowerShell cmdlets (from the NetTCPIP/NetAdapter modules) for inspecting network configuration — the structured-object replacement for `ipconfig`.
    - **Get-NetIPConfiguration** = a high-level summary similar to `ipconfig`, showing IP address, gateway, and DNS servers per interface.
    - **Get-NetIPAddress** = returns detailed IP address assignment objects, filterable by `-InterfaceAlias`, `-AddressFamily` (IPv4/IPv6), etc.
    - **Get-NetAdapter** = lists network adapters and their link status (Up/Down), speed, and MAC address — the structured-object equivalent of `Get-NetIPConfiguration`'s adapter-level view.
    - **New-NetIPAddress / Set-NetIPInterface / Set-DnsClientServerAddress** = the modern PowerShell cmdlets for configuring a static IP, gateway, and DNS servers on an interface, replacing the old `netsh interface ip set address` syntax.
- **Invoke-WebRequest** = sends an HTTP/HTTPS request and returns the full response (status code, headers, raw content) — the PowerShell equivalent of `curl`, useful for testing web endpoints or scraping simple HTML content.
    - **-Uri** = the target URL.
    - **-Method** = the HTTP method (GET, POST, PUT, DELETE, etc.).
    - **-Headers** = a hashtable of custom HTTP headers to send (commonly used for API keys/auth tokens).
    - **-Body** = the request body content, often a JSON string for a POST/PUT to a REST API.
    - **-OutFile** = saves the response content directly to a file instead of returning it to the pipeline (the equivalent of `curl -o`).
    - **-UseBasicParsing** = (older PowerShell versions only) skips using the Internet Explorer HTML parsing engine, avoiding a common error on servers without IE components installed. Not needed in PowerShell 7+.
- **Invoke-RestMethod** = similar to `Invoke-WebRequest`, but specifically designed for REST APIs — it automatically parses a JSON or XML response body directly into PowerShell objects, rather than returning raw text you'd have to parse yourself. This is the cmdlet you'll use constantly when scripting against Microsoft Graph or any other REST API directly (rather than through a dedicated SDK module).

```
$Headers = @{ Authorization = "Bearer $Token" }
Invoke-RestMethod -Uri "https://graph.microsoft.com/v1.0/me" -Headers $Headers -Method
Get
```

- **Invoke-Command** = executes a script block on one or more remote computers using PowerShell Remoting (WinRM) — this is the foundational remoting cmdlet, roughly analogous in purpose to running a command over SSH on a remote Linux box, but built on Windows Remote Management instead.

```
Invoke-Command -ComputerName LAB-SRV01,LAB-SRV02 -ScriptBlock { Get-Service -Name
Spooler }
```

    - **-ComputerName** = accepts one or more remote computer names to run the command against, in parallel.
    - **-Credential** = specifies alternate credentials to authenticate with on the remote machine(s).
    - **-ScriptBlock** = the code to execute remotely.
    - **-AsJob** = runs the remote command as a background job instead of waiting synchronously for it to complete, letting the script continue immediately and check back on results later.
    - **-FilePath** = runs an entire local .ps1 script file remotely instead of an inline script block.
- **Enter-PSSession / New-PSSession / Exit-PSSession** = PowerShell Remoting cmdlets for interactive and persistent remote sessions.
    - **Enter-PSSession -ComputerName \<name\>** = opens an *interactive* remote session, where your prompt changes to reflect you're now working directly on the remote machine — the closest PowerShell equivalent to `ssh user@host`.
    - **New-PSSession** = creates a *persistent, reusable* remote session object (stored in a variable) that keeps state (like variables or imported modules) between multiple separate `Invoke-Command` calls, instead of reconnecting fresh each time.
    - **Exit-PSSession** = leaves an interactive remote session and returns to the local prompt (equivalent to typing `exit` over SSH).
- **Enable-PSRemoting** = configures the local machine to accept incoming PowerShell Remoting connections — configures the WinRM service, sets it to start automatically, and creates the necessary firewall exceptions. Must be run once (as admin) on any machine you intend to remotely manage via Invoke-Command/Enter-PSSession.
    - **-Force** = suppresses the interactive confirmation prompts, useful for running unattended during initial server provisioning/imaging.
- **Start-Job / Get-Job / Receive-Job / Wait-Job / Stop-Job** = PowerShell's native background job system, letting a script kick off long-running work in a separate process and check back on it later instead of blocking — analogous in purpose to backgrounding a task with `&` in Bash, but with far more structure around tracking state and retrieving results.
    - **Start-Job -ScriptBlock { ... }** = starts a script block running as a background job and immediately returns control to the calling script.
    - **Get-Job** = lists all jobs in the current session and their state (Running, Completed, Failed).
    - **Receive-Job -Id \<id\>** = retrieves the output/results produced so far by a job (add `-Wait` to block until it finishes, or `-Keep` to be able to retrieve the results again later).
    - **Wait-Job** = blocks script execution until the specified job(s) reach a completed state.
    - **Stop-Job / Remove-Job** = stops a running job, or removes a completed job's record from memory.
- **Register-ScheduledTask / Get-ScheduledTask** = the modern PowerShell cmdlet family for creating and managing Windows Task Scheduler jobs — the structured-object replacement for the legacy `schtasks.exe` command-line tool.

```
$Action  = New-ScheduledTaskAction -Execute "powershell.exe" -Argument "-File
C:\Scripts\CleanTemp.ps1"
$Trigger = New-ScheduledTaskTrigger -Daily -At "2:00AM"
Register-ScheduledTask -TaskName "Nightly Temp Cleanup" -Action $Action -Trigger
$Trigger -User "SYSTEM"
```

    - **New-ScheduledTaskAction** = defines what the task actually runs (a program/script and its arguments).
    - **New-ScheduledTaskTrigger** = defines when the task runs (`-Daily`, `-Weekly`, `-Once`, `-AtStartup`, `-AtLogOn`).
    - **Get-ScheduledTask** = lists existing scheduled tasks and their current state.
    - **Start-ScheduledTask / Disable-ScheduledTask / Unregister-ScheduledTask** = manually trigger, disable, or delete a scheduled task.

- **Get-ExecutionPolicy / Set-ExecutionPolicy** = controls whether PowerShell will allow local .ps1 scripts to run at all — a safety mechanism against accidentally (or maliciously) running untrusted scripts, not a genuine security boundary against a determined attacker.
    - **Restricted** = (the historical Windows client default) no scripts can run at all, only interactive commands.
    - **AllSigned** = only scripts signed by a trusted publisher can run.
    - **RemoteSigned** = local scripts you wrote yourself can run freely, but scripts downloaded from the internet must be digitally signed (this is the standard, sensible default for most sysadmin workstations).
    - **Unrestricted** = all scripts run, but downloaded scripts still show a warning prompt first.
    - **Bypass** = nothing is blocked and no warnings are shown at all — typically used only for a specific automated/unattended process, not as a general workstation policy.
    - **-Scope** = execution policy can be set at different scopes (Process, CurrentUser, LocalMachine); Process scope only affects the current PowerShell window and is a common safe way to temporarily allow a script to run without changing the system-wide policy.
- **Get-Acl / Set-Acl** = reads or writes the NTFS security descriptor (owner, permissions, auditing) of a file, folder, or registry key — the PowerShell-native equivalent of `icacls`, returning a structured object you can inspect or copy between items rather than parsing text output.

```
$Acl = Get-Acl -Path "C:\Shares\Department"
$Acl.Access  # lists each individual access rule
Set-Acl -Path "C:\Shares\NewDepartment" -AclObject $Acl   # copies the entire ACL to
another folder
```

- **Get-FileHash** = computes a cryptographic hash of a file — the PowerShell equivalent of Linux's `sha256sum`/`md5sum`, most commonly used to verify a downloaded file matches a vendor-published checksum.
    - **-Algorithm** = specifies the hash algorithm to use (SHA256 is the modern default; MD5 and SHA1 are also supported for compatibility but considered cryptographically weak).
- **Get-Random** = generates a random number, or picks a random item from an input collection — used for things like jittering scheduled task start times across a fleet of machines to avoid a 'thundering herd' effect all hitting a server at once.
    - **-Minimum / -Maximum** = bounds for the generated number.
    - **-InputObject** = when piped a collection, returns one (or more, with `-Count`) randomly selected item(s) from it instead of a random number.
- **Start-Sleep** = pauses script execution for a specified duration — the PowerShell equivalent of Bash's `sleep`.
    - **-Seconds \<n\>** or **-Milliseconds \<n\>** = specifies how long to pause.
- **Get-Date** = returns the current date/time as a rich .NET DateTime object (not just a formatted text string), which is what allows date arithmetic like `(Get-Date).AddDays(-7)` to work directly.
    - **-Format \<string\>** = formats the output as a specific text string pattern (ex. `-Format "yyyy-MM-dd"`).
    - **-UFormat** = uses Unix-style format specifiers instead, for compatibility with cross-platform scripting conventions.
- **Out-File / Out-GridView / Out-Null** = cmdlets that send pipeline output to a destination other than the console.
    - **Out-File** = writes output to a text file, functionally very similar to `Set-Content` but designed specifically for formatted console-style output rather than raw string data.
    - **Out-GridView** = opens pipeline output in an interactive, sortable/filterable graphical grid window — an extremely handy tool for exploring large result sets visually during ad-hoc troubleshooting (Windows PowerShell 5.1 and PowerShell 7+ with the Microsoft.PowerShell.GraphicalTools module).
    - **Out-Null** = discards output entirely, sending it nowhere — used to suppress a cmdlet's return value in a script when you only care about its side effect, not its output.

- **Write-Output / Write-Host / Write-Verbose / Write-Warning / Write-Error** = the family of cmdlets for producing script output and diagnostic messages, each writing to a fundamentally different PowerShell output stream — mixing these up is one of the most common scripting mistakes.
    - **Write-Output** = writes an object to the standard **output stream** (stream 1) — this is the actual pipeline data other commands can capture and process further.
    - **Write-Host** = writes text directly to the console/host display and nothing else — it does **not** go into the pipeline, so `$result = MyFunction` will never capture something written with `Write-Host`. Use this only for direct human-facing console messages, never for data you intend to return from a function.
    - **Write-Verbose** = writes to the verbose stream, only visible when `-Verbose` is specified when calling the command (or `$VerbosePreference` is set).
    - **Write-Warning** = writes a yellow warning message to the warning stream.
    - **Write-Error** = writes to the error stream (stream 2) — this produces a *non-terminating* error by default, meaning the script keeps running afterward unless `-ErrorAction Stop` is also specified.
    - **Write-Debug** = writes to the debug stream, only visible when `-Debug` is specified or `$DebugPreference` is set — useful for verbose internal diagnostic detail you don't want cluttering normal output.
- **Get-ComputerInfo** = returns a single, comprehensive object summarizing the local machine's OS version/build, BIOS, hardware, hotfix, and network configuration — the PowerShell-native, scriptable replacement for the legacy `systeminfo` command, returning structured properties instead of plain text.

```
Get-ComputerInfo | Select-Object CsName, OsName, OsVersion, WindowsBuildLabEx,
BiosSMBIOSBIOSVersion
```

- **Rename-Computer** = renames the local (or a remote) computer — requires a restart to fully take effect.
    - **-NewName** = the new computer name.
    - **-DomainCredential** = required if the computer is domain-joined, since renaming a domain-joined computer also requires updating its AD computer object.
    - **-Restart** = automatically restarts the computer immediately after the rename to complete the process.
- **Add-Computer / Remove-Computer** = joins or unjoins the local computer to/from an Active Directory domain — the PowerShell-native equivalent of the Linux `realm join`/`realm leave` workflow.

```
Add-Computer -DomainName contoso.com -Credential (Get-Credential) -Restart
```

    - **-OUPath** = specifies a target Organizational Unit's distinguished name to place the new computer object in, instead of the default Computers container.
    - **-Credential** = requires an account with sufficient rights to create/join computer objects in the target domain.
    - **Remove-Computer -UnjoinDomainCredential** = leaves the current domain and returns the machine to a workgroup, requiring credentials with rights to remove the computer object.
- **Test-ComputerSecureChannel / Repair-ComputerSecureChannel** = checks (and optionally repairs) the secure channel trust relationship between a domain-joined computer and a domain controller — the PowerShell-native equivalent of the Windows CLI `nltest /sc_query` and `/sc_reset` combination covered earlier, used for the classic 'trust relationship between this workstation and the primary domain failed' error.

```
Test-ComputerSecureChannel -Repair -Credential (Get-Credential)
```

- **Restart-Computer / Stop-Computer** = restart or shut down the local (or one/more remote) computers — the PowerShell-native equivalents of the legacy `shutdown` command.
    - **-ComputerName** = accepts an array, letting you restart/shut down multiple remote machines in a single command.
    - **-Force** = forces the action even if applications with unsaved data are open.

    - **-Wait** = pauses the script until the target machine has finished restarting and is reachable again — useful for orchestrating a scripted patch-and-reboot sequence.
- **Get-HotFix** = lists installed Windows updates/hotfixes on the local (or remote) machine — a scriptable, structured-object alternative to checking Control Panel's installed updates list, commonly used to verify whether a specific KB has been successfully deployed across a fleet.

```
Get-HotFix -Id KB5044284
Get-HotFix -ComputerName LAB-SRV01,LAB-SRV02 | Sort-Object InstalledOn -Descending
```

- **Get-LocalUser / New-LocalUser / Set-LocalUser / Remove-LocalUser** = manage **local** (non-domain) user accounts on the machine — the PowerShell-native replacement for the legacy `net user` command, part of the built-in Microsoft.PowerShell.LocalAccounts module.

```
New-LocalUser -Name "svc_example" -Password (ConvertTo-SecureString "Example#Pass3"
-AsPlainText -Force) -PasswordNeverExpires
Set-LocalUser -Name "svc_example" -AccountNeverExpires
```

    - **Get-LocalUser** = lists all local accounts and their enabled/disabled status.
    - **Disable-LocalUser / Enable-LocalUser** = toggle whether a local account can log in.
- **Get-LocalGroup / New-LocalGroup / Add-LocalGroupMember / Remove-LocalGroupMember** = manage local security groups (most commonly, the local Administrators or Remote Desktop Users groups) — the PowerShell-native replacement for `net localgroup`.

```
Add-LocalGroupMember -Group "Administrators" -Member "CONTOSO\asample"
```

    - **-Member** = accepts local account names, or domain accounts in `DOMAIN\username` format, allowing you to grant a domain user local admin rights on a specific machine.
- **Group Policy Module (GroupPolicy) — Get-GPO / New-GPO / Set-GPRegistryValue** = manages Group Policy Objects (GPOs) directly from PowerShell, part of RSAT alongside the ActiveDirectory module — used for scripting policy creation, linking, and reporting instead of navigating the Group Policy Management Console (GPMC) GUI.

```
New-GPO -Name "Disable Telnet Client" | New-GPLink -Target
"OU=Workstations,DC=contoso,DC=com"
Set-GPRegistryValue -Name "Disable Telnet Client" -Key
"HKLM\SOFTWARE\Policies\Microsoft\Telnet" -ValueName "Disabled" -Type DWord -Value 1
```

    - **Get-GPO -All** = lists every GPO in the domain.
    - **New-GPLink** = links an existing GPO to a specific OU, domain, or site.
    - **Set-GPRegistryValue** = the most common way to script a registry-based policy setting directly into a GPO without touching the GPMC editor.
    - **Get-GPResultantSetOfPolicy** = generates an RSoP report for a specific user/computer — the PowerShell-scriptable equivalent of running `gpresult /h`.
    - **Backup-GPO / Import-GPO** = back up a GPO to disk and restore/import it elsewhere, useful for migrating policies between environments or maintaining version history outside of GPMC.
- **WSMan Configuration (Get-Item WSMan:\localhost\... / Set-WSManInstance)** = PowerShell Remoting (WinRM) exposes its own configuration through a dedicated `WSMan:` PSDrive, letting you inspect and tune remoting-specific settings (max concurrent connections, timeouts, trusted hosts list) the same way you'd browse a filesystem.

```
Get-Item WSMan:\localhost\Client\TrustedHosts
Set-Item WSMan:\localhost\Client\TrustedHosts -Value "*.contoso.com" -Force
```

    - **TrustedHosts** = required when connecting via PowerShell Remoting to a target machine using only an IP address (rather than a resolvable hostname) or across a non-domain trust boundary, since Kerberos can't validate the connection by name in those cases.

- **Get-PSSessionConfiguration / Register-PSSessionConfiguration (JEA)** = Just Enough Administration (JEA) is a PowerShell Remoting security model that lets you grant a specific user or group access to run **only** a tightly-defined, pre-approved set of cmdlets/functions in a restricted remote session — instead of full administrative access. This is the PowerShell-native equivalent in spirit to Linux's `sudo` with a carefully scoped `/etc/sudoers` entry limiting exactly which commands are allowed.
    - **Register-PSSessionConfiguration -Name \<name\> -Path \<RoleCapabilityOrSessionConfigFile\>** = registers a new constrained endpoint that users connect to instead of a normal unrestricted PowerShell session.
    - A **Role Capability file** (.psrc) defines exactly which cmdlets, functions, and parameters are permitted inside that constrained session — anything not explicitly allowed is simply unavailable to the connecting user, even if they otherwise have local admin rights.
- **Get-Credential and PSCredential objects, revisited** = expanding on the earlier mention: a `PSCredential` object bundles a username and a `SecureString` password together, and is the expected input type for the `-Credential` parameter found on dozens of cmdlets across nearly every module (remoting, AD, Azure app-only auth flows, mapped drives). Building one non-interactively for automation requires combining `New-Object System.Management.Automation.PSCredential` with a securely-sourced password (never a bare plaintext string committed to a script).

```
$SecurePwd = Get-Content "C:\Secure\svc_example.pw" | ConvertTo-SecureString
$Cred = New-Object System.Management.Automation.PSCredential("CONTOSO\svc_example",
$SecurePwd)
```

- **Get-Counter** = reads live Windows Performance Counter data directly from PowerShell — the scriptable equivalent of the classic `perfmon`/`typeperf` tools, useful for capturing a quick CPU/memory/disk performance snapshot inside a script without needing to configure a full Data Collector Set.

```
Get-Counter -Counter "\Processor(_Total)\% Processor Time" -SampleInterval 2
-MaxSamples 5
```

- **Get-Uptime / (Get-CimInstance Win32_OperatingSystem).LastBootUpTime** = there's no single dedicated `Get-Uptime` cmdlet built into Windows PowerShell, but the standard idiom for calculating system uptime is subtracting a machine's last boot time (from CIM) from the current time — worth knowing since it comes up constantly in health-check scripts.

```
(Get-Date) - (Get-CimInstance Win32_OperatingSystem).LastBootUpTime
```

- **Get-WindowsFeature / Install-WindowsFeature / Remove-WindowsFeature** = manage Windows Server Roles and Features (the ServerManager module) — the PowerShell-native way to install server roles like DNS, DHCP, Active Directory Domain Services, or IIS without touching the Server Manager GUI, essential for scripting unattended server builds/imaging.

```
Get-WindowsFeature | Where-Object Installed
Install-WindowsFeature -Name AD-Domain-Services -IncludeManagementTools
Install-WindowsFeature -Name DNS,DHCP
```

    - **-IncludeManagementTools** = also installs the associated RSAT management tools/cmdlets for the role (e.g., installing AD-Domain-Services without this flag installs the DC role binaries but not the ActiveDirectory PowerShell module itself).
    - **-Restart** = automatically restarts the server if the feature installation requires it.
- **Get-WindowsOptionalFeature / Enable-WindowsOptionalFeature** = manages optional features specifically on **Windows client** editions (Windows 10/11) — such as enabling the legacy Telnet Client, Hyper-V, or Windows Subsystem for Linux — the client-OS counterpart to `Install-WindowsFeature` on Server.

```
Enable-WindowsOptionalFeature -Online -FeatureName TelnetClient -All
```

    - **-Online** = applies the change to the currently running OS instance (as opposed to `-Path`, which can service an offline mounted image instead).

- **Common Cmdlet Aliases** = PowerShell ships with many built-in aliases (short, often Unix-inspired shortcuts) for its most common cmdlets, useful for fast interactive typing but generally discouraged inside saved scripts (where the full cmdlet name is far more readable to someone else later). Worth recognizing when reading other people's one-liners.
  - **gci, ls, dir** → Get-ChildItem
  - **gc, cat, type** → Get-Content
  - **gm** → Get-Member
  - **gps, ps** → Get-Process
  - **gsv** → Get-Service
  - **?, where** → Where-Object
  - **%, foreach** → ForEach-Object
  - **select** → Select-Object
  - **sort** → Sort-Object
  - **sls** → Select-String
  - **iwr, curl, wget** → Invoke-WebRequest
  - **icm** → Invoke-Command
  - **kill** → Stop-Process
  - **cp, copy, cpi** → Copy-Item
  - **mv, move, mi** → Move-Item
  - **rm, del, ri** → Remove-Item
  - **cd, chdir** → Set-Location
  - **pwd** → Get-Location
  - **echo, write** → Write-Output
  - **cls, clear** → Clear-Host
  - **Get-Alias** = lists every currently defined alias in the session; **Set-Alias** = creates your own custom alias for any command.
- **ForEach-Object -Parallel / Start-ThreadJob (PowerShell 7+)** = PowerShell 7 added native support for running pipeline iterations concurrently across multiple threads, dramatically speeding up tasks like pinging or querying hundreds of servers, without needing to hand-roll runspace pools yourself (the way advanced scripters had to on Windows PowerShell 5.1).

```
$Servers | ForEach-Object -Parallel {
    Test-Connection -ComputerName $_ -Count 1 -Quiet
} -ThrottleLimit 10
```

  - **-ThrottleLimit** = caps the maximum number of concurrent threads running at once, preventing the local machine (or the target servers/network) from being overwhelmed.
  - **-AsJob** = runs the entire parallel operation as a single background job instead of blocking the console until all iterations finish.
  - **Start-ThreadJob** = a lighter-weight alternative to `Start-Job` for background work, using threads within the same process instead of spawning an entirely separate PowerShell process per job (much faster to start, lower memory overhead) — installed from the PowerShell Gallery on Windows PowerShell 5.1, built-in by default in PowerShell 7+.
- **Regex Quick Reference (for -match, -replace, Select-String)** = a condensed cheat-sheet of the most commonly needed .NET regular expression tokens used across `-match`, `-replace`, `-split`, and `Select-String` throughout this guide.
  - **^** = anchors the match to the start of the string/line.

    - **$** = anchors the match to the end of the string/line.
    - **.** = matches any single character.
    - **\\d** = matches any digit (0-9); **\\D** = any non-digit.
    - **\\w** = matches any word character (letters, digits, underscore); **\\W** = any non-word character.
    - **\\s** = matches any whitespace character; **\\S** = any non-whitespace character.
    - ***** = zero or more of the preceding token; **+** = one or more; **?** = zero or one.
    - **{n,m}** = matches between n and m repetitions of the preceding token.
    - **[abc]** = matches any single character within the set; **[^abc]** = matches any character NOT in the set.
    - **(...)** = a capturing group — its matched value becomes available afterward in the `$Matches` automatic variable following a successful `-match`.
    - **\|** = alternation (OR) between two patterns.
- **Set-Location / Get-Location / Push-Location / Pop-Location** = change or query the current working directory — the PowerShell equivalents of `cd` and `pwd`. `Push-Location` saves the current directory onto a stack before moving, so a later `Pop-Location` jumps straight back. Alias: `cd`, `sl`, `pwd`, `gl`, `pushd`, `popd`.
- **Clear-Host** = clears the console screen. Alias: `cls`, `clear`.
- **Get-Alias / Set-Alias / New-Alias** = manage PowerShell's built-in command aliases, which is why `ls`, `cat`, `pwd`, and `rm` all work in PowerShell even though the real cmdlets are `Get-ChildItem`, `Get-Content`, etc. `Get-Alias ls` reveals what an alias actually points to; `New-Alias` defines your own.
- **Read-Host** = pauses a script and prompts the user to type a value interactively, optionally masking the input for a password. (ex. `$Cred = Read-Host -AsSecureString "Enter password"`)
- **Join-Path / Split-Path / Resolve-Path** = build and take apart filesystem paths correctly, without manually concatenating strings and backslashes.
    - **Join-Path** = safely combines a parent path and a child path into one, handling the separating slash for you. (ex. `Join-Path $Root "logs"`)
    - **Split-Path** = extracts just the parent directory (`-Parent`), the leaf filename (`-Leaf`), or the qualifying drive (`-Qualifier`) from a full path.
    - **Resolve-Path** = converts a relative path into its full, absolute path, expanding `..`/`.` and verifying the path actually exists.
- **Get-Clipboard / Set-Clipboard** = read from or write to the Windows clipboard directly from a script — useful for quickly grabbing a value or piping command output straight to paste elsewhere.
- **Start-Transcript / Stop-Transcript** = records every command and its output in a PowerShell session to a text file, from the moment you start it until you stop it — invaluable for producing an audit trail of an interactive troubleshooting session.
- **Get-NetTCPConnection** = lists active TCP connections and their state (Listen, Established, etc.) with the owning process ID — the modern PowerShell replacement for parsing `netstat -ano` output. (ex. `Get-NetTCPConnection -LocalPort 443`)
- **Get-DnsClientCache / Clear-DnsClientCache** = view or flush the local DNS resolver cache — the cmdlet equivalents of `ipconfig /displaydns` and `ipconfig /flushdns`.
- **Invoke-Item** = opens a file or folder with whatever application is registered as its default handler, exactly like double-clicking it in File Explorer. (ex. `Invoke-Item .\report.pdf`) Alias: `ii`.
- **Write-Progress** = displays a progress bar in the console during a long-running loop, showing the user the script hasn't hung. (ex. inside a `foreach`, `Write-Progress -Activity "Processing" -Status $Item -PercentComplete $Pct`)
- **Get-History / Invoke-History** = `Get-History` lists the commands run so far in the current session (alias `h`); `Invoke-History` re-runs a specific command from that list by its ID, similar to Bash's `!<n>`.
- **New-Guid** = generates a new, random globally-unique identifier (GUID) — commonly used to generate unique names, correlation IDs, or test data.

- **Unregister-ScheduledTask / Start-ScheduledTask** = complete the scheduled-task lifecycle alongside `Register-ScheduledTask`/`Get-ScheduledTask` covered above — `Start-ScheduledTask` triggers a defined task to run immediately (outside its normal trigger), and `Unregister-ScheduledTask` deletes it entirely.
- **Wait-Process** = pauses script execution until a specific running process exits, which is useful for scripting an installer or external tool that must finish before the next step runs.

Common Error Messages & Troubleshooting Quick Reference

A quick-reference of PowerShell and Windows sysadmin error messages/behaviors you'll run into constantly, what they actually mean, and the standard first fix to try — the PowerShell equivalent of the 'Exam Scenario' callouts throughout the Linux guide.

- **"The term '...' is not recognized as the name of a cmdlet..."** = PowerShell can't find a command matching that name in any currently loaded module or the system PATH.
  - Check for a typo first — this is by far the most common cause.
  - If it's a module cmdlet, confirm the module is actually installed and imported: `Get-Module -ListAvailable \<ModuleName\>`, then `Import-Module \<ModuleName\>`.
  - If it's meant to be an external .exe, confirm it's actually in `$env:Path`, or call it with its full file path.
- **"Access is denied" / "Insufficient privileges to complete the operation"** = a permissions error, but the *source* of the required permission differs dramatically depending on which type of cmdlet you're running.
  - For local system operations (services, files, registry): re-run PowerShell **elevated** (Run as Administrator).
  - For ActiveDirectory cmdlets: your account lacks the required delegated AD permission over that specific OU/object — this is not fixed by running PowerShell as local admin, since AD permissions are entirely separate from local machine admin rights.
  - For Microsoft.Graph cmdlets: your `Connect-MgGraph -Scopes` request didn't include a scope broad enough for the operation, **or** an admin has not granted consent for that scope in the tenant — check with `Get-MgContext` first.
  - For Az cmdlets: your account (or the connected service principal) lacks the required Azure RBAC role at that scope — check with `Get-AzRoleAssignment`.
- **"Cannot bind argument to parameter '...' because it is null"** = a mandatory parameter received `$null` instead of a real value — extremely common when a variable you expected to be populated from an earlier command actually came back empty (e.g., a `Get-ADUser` lookup that found no match).
  - Add a `[ValidateNotNullOrEmpty()]` attribute (see Advanced Functions) to catch this earlier and with a clearer error, or add explicit `if ($null -eq $Variable)` checks before using a value that came from an earlier lookup.
- **"The response has been terminated because the request has exceeded the response timeout" (Graph/REST)** = a network-level or throttling timeout on a REST call — first confirm it isn't actually an HTTP 429 throttling response (see the Batching/Throttling entry in the Graph section) being reported generically, then check outbound firewall/proxy rules to the Graph endpoint.
- **"WinRM cannot process the request... Access is denied" (Remoting)** = the classic PowerShell Remoting connection failure, usually one of three causes.
  - The target machine doesn't have `Enable-PSRemoting` configured at all — run it there first.
  - You're connecting by IP address rather than resolvable hostname across a non-domain boundary — add the target to `WSMan:\localhost\Client\TrustedHosts` on the client.
  - A firewall is blocking the WinRM port (5985 for HTTP, 5986 for HTTPS) between the two machines.
- **"The trust relationship between this workstation and the primary domain failed"** = the computer's local secure channel password no longer matches what AD has on record for that computer account — usually caused by restoring a VM from an old snapshot/backup after the computer account's password had already rotated.

    - Fix with `Test-ComputerSecureChannel -Repair -Credential (Get-Credential)` (requires domain admin-equivalent rights), or as a last resort, unjoin and rejoin the domain entirely.
- **"Execution of scripts is disabled on this system"** = the current Execution Policy is blocking the script from running (see the Get-ExecutionPolicy / Set-ExecutionPolicy entry) — most commonly fixed for a single session with `Set-ExecutionPolicy -Scope Process -ExecutionPolicy Bypass`, without changing the machine-wide policy.
- **A cmdlet returns unexpectedly few results (silently truncated)** = several modules (Exchange Online, Microsoft.Graph, some AD queries) impose a default page size/result cap that silently truncates large result sets rather than throwing an error.
    - Exchange Online: add `-ResultSize Unlimited`.
    - Microsoft.Graph: add the `-All` switch.
    - Always check a cmdlet's help/parameters for a page-size or result-limit parameter before assuming a query genuinely found nothing more.
- **"Get-WmiObject : Invalid namespace" or DCOM connection errors (remote WMI)** = usually a firewall or legacy-protocol issue specific to WMI's older DCOM transport — the most reliable fix is simply switching the script to `Get-CimInstance` instead, which uses the modern WSMan/WinRM transport and is far less commonly blocked by current firewall configurations.
- **A variable seems to "disappear" after a function or ForEach-Object block ends** = classic scope confusion (see the Scope entry in Scripting Fundamentals) — a variable set inside a function or script block is local to that block by default and doesn't leak out, unlike Bash. Use `$script:` or `$global:` scope modifiers if the variable genuinely needs to be visible outside the block it was set in.
- **try/catch doesn't catch an error at all** = the single most common `try/catch` mistake: the cmdlet inside the `try` block produced a **non-terminating** error (the default for most cmdlets), which `catch` cannot intercept. Add `-ErrorAction Stop` to the specific command inside the `try` block to force it to become a terminating error that `catch` can actually handle.
- **A scheduled task shows "Last Run Result: 0x1" or similar hex code with no other detail** = the task technically ran but the underlying script/command returned a non-zero exit code — check `$LASTEXITCODE` behavior for any native .exe calls inside the script, and confirm the task's configured 'Run whether user is logged on or not' and working-directory settings match what the script expects (a script referencing relative paths often fails silently when run by Task Scheduler under a different working directory than expected).
- **AD replication shows a large "USN" gap or a DC hasn't replicated in days** = check `Get-ADReplicationFailure` and `repadmin /showrepl` first — common root causes include a firewall blocking AD replication ports, DNS resolution failures between sites, or a domain controller that was restored from an old backup/snapshot without the appropriate non-authoritative restore precautions (directly relevant to any DC migration/backup planning work).

Modules

Active Directory Module (ActiveDirectory)

The ActiveDirectory module ships with the Remote Server Administration Tools (RSAT) and is the primary way sysadmins manage on-premises AD DS objects from PowerShell instead of the ADUC (Active Directory Users and Computers) GUI. Nearly every cmdlet here follows the same Verb-AD<Noun> naming pattern and shares a common set of filtering parameters.

- **Get-ADUser** = retrieves one or more Active Directory user account objects and their attributes.

```
Get-ADUser -Identity asample -Properties *
Get-ADUser -Filter "Department -eq 'IT'" -Properties Department, Title
```

    - **-Identity** = specifies a single user by SamAccountName, DistinguishedName, GUID, or SID.

    - **-Filter** = a required parameter for searching multiple users, using PowerShell-style comparison syntax (`-eq`, `-like`, `-and`, `-or`) rather than raw LDAP syntax — this is the AD module's biggest advantage over raw `ldapsearch`.
    - **-LDAPFilter** = an alternative to `-Filter` that accepts raw LDAP query syntax directly, useful when porting an existing LDAP filter or needing a capability `-Filter` doesn't expose.
    - **-Properties** = by default, Get-ADUser only returns a small subset of common attributes; `-Properties *` (or a specific comma-separated list) is required to retrieve additional attributes like `Manager`, `Department`, or custom extension attributes.
    - **-SearchBase** = restricts the search to a specific OU's distinguished name, instead of searching the entire domain.
    - **-SearchScope** = controls how deep the search goes relative to `-SearchBase` (Base, OneLevel, or Subtree).
- **New-ADUser** = creates a new Active Directory user account.

```
New-ADUser -Name "Alex Sample" -SamAccountName asample -UserPrincipalName
asample@contoso.com `
  -Path "OU=Users,OU=Corporate,DC=contoso,DC=com" -AccountPassword
(ConvertTo-SecureString "Example#Pass1" -AsPlainText -Force) `
  -Enabled $true -ChangePasswordAtLogon $true
```

    - **-Path** = the distinguished name of the OU the new account should be created in.
    - **-AccountPassword** = requires a `SecureString`, not plain text — must be built with `ConvertTo-SecureString -AsPlainText -Force`.
    - **-Enabled** = new accounts are disabled by default unless this is explicitly set to `$true`.
    - **-ChangePasswordAtLogon** = forces the user to set their own password on first login.
- **Set-ADUser** = modifies attributes on an existing user account.

```
Set-ADUser -Identity asample -Department "Example Department" -Title "Job Title
Example"
```

    - **-Replace / -Add / -Clear** = for attributes that don't have a dedicated named parameter, these generic hashtable-based parameters let you set, append to, or clear any arbitrary AD attribute (ex. `-Replace @{extensionAttribute1="Contractor"}`).
- **Remove-ADUser / Disable-ADAccount / Enable-ADAccount** = deletes, disables, or re-enables an account, respectively — disabling rather than deleting is the standard practice for offboarding, since it preserves the account's SID and group memberships for auditing/eDiscovery purposes.
    - **-Confirm:$false** = commonly appended to suppress the interactive confirmation prompt in automated offboarding scripts.
- **Unlock-ADAccount / Get-ADUser -Properties LockedOut** = unlocks a user account that has been locked out due to too many failed password attempts (analogous to Linux's `faillock --reset`).

```
Unlock-ADAccount -Identity asample
Search-ADAccount -LockedOut | Select-Object Name, SamAccountName
```

- **Set-ADAccountPassword / Reset password flow** = resets a user's password from the command line.

```
Set-ADAccountPassword -Identity asample -Reset -NewPassword (ConvertTo-SecureString
"Example#Pass2" -AsPlainText -Force)
```

    - Almost always paired with `Set-ADUser -Identity asample -ChangePasswordAtLogon $true` so the temporary password must be changed on next login.
- **Search-ADAccount** = a purpose-built cmdlet for finding accounts matching a specific state, without needing to write a raw `-Filter` string yourself.
    - **-LockedOut** = finds all currently locked-out accounts.

    - **-AccountDisabled** = finds all disabled accounts.
    - **-PasswordExpired** = finds accounts whose password has already expired.
    - **-AccountExpiring -TimeSpan \<span\>** = finds accounts set to expire within a given time window — useful for a scheduled report flagging upcoming contractor account expirations.
    - **-UsersOnly / -ComputersOnly** = restricts results to only user or only computer accounts.
- **Get-ADGroup / New-ADGroup** = retrieves or creates Active Directory security or distribution groups.

```
New-ADGroup -Name "Contoso-Helpdesk" -GroupScope Global -GroupCategory Security -Path
"OU=Groups,DC=contoso,DC=com"
```

    - **-GroupScope** = DomainLocal, Global, or Universal — determines where the group can be used and which members it can contain across trusts/domains.
    - **-GroupCategory** = Security (used for permissions/access control) or Distribution (used only for email distribution lists, no security token).
- **Get-ADGroupMember / Add-ADGroupMember / Remove-ADGroupMember** = manage group membership.

```
Get-ADGroupMember -Identity "Contoso-Helpdesk" -Recursive
Add-ADGroupMember -Identity "Contoso-Helpdesk" -Members asample,cdemo
Remove-ADGroupMember -Identity "Contoso-Helpdesk" -Members asample -Confirm:$false
```

    - **-Recursive** = on `Get-ADGroupMember`, expands nested group membership so you see every actual user, not just the immediate direct members (which may themselves be groups).
- **Get-ADPrincipalGroupMembership** = the reverse lookup of `Get-ADGroupMember` — shows every group a specific user (or computer) is a member of, rather than showing a group's members.

```
Get-ADPrincipalGroupMembership -Identity asample | Select-Object Name
```

- **Get-ADComputer / New-ADComputer** = retrieves or creates computer account objects in Active Directory — used heavily when scripting bulk pre-staging of computer accounts before an imaging deployment.
    - **-Filter** and **-Properties** behave the same way as on `Get-ADUser`.
    - **-Enabled** = new computer accounts default to enabled, but this can still be explicitly controlled.
- **Get-ADOrganizationalUnit / New-ADOrganizationalUnit** = retrieves or creates Organizational Units (OUs) — the containers used to organize users/computers/groups and apply Group Policy at a granular level.
    - **-ProtectedFromAccidentalDeletion** = a Boolean flag (defaults to `$true` on new OUs) that prevents the OU from being deleted without first explicitly disabling this protection — a safeguard against a misclick wiping out an entire OU structure.
- **Move-ADObject** = moves any AD object (user, computer, group, OU) from one container/OU to another, identified by distinguished name.

```
Move-ADObject -Identity "CN=Alex Sample,OU=Users,DC=contoso,DC=com" -TargetPath
"OU=Contractors,DC=contoso,DC=com"
```

- **Get-ADDomain / Get-ADForest / Get-ADDomainController** = retrieve high-level information about the AD domain, forest, and domain controllers themselves.
    - **Get-ADDomain** = shows domain-level configuration: NetBIOS name, domain functional level, PDC emulator, infrastructure master, and other FSMO role holders.
    - **Get-ADForest** = shows forest-wide configuration: forest functional level, schema master, domain naming master, and all domains in the forest.
    - **Get-ADDomainController -Filter *** = lists all domain controllers in the domain, along with their site, IP address, and which FSMO roles (if any) each one holds.
    - **Get-ADDomainController -Discover** = queries DNS SRV records to locate the nearest available domain controller, similar in purpose to Linux's `realm discover` or `nltest /dsgetdc:`.

- **Get-ADReplicationPartnerMetadata / Get-ADReplicationFailure** = PowerShell-native cmdlets for checking Active Directory replication health between domain controllers — a scriptable alternative to running `repadmin` directly.

```
Get-ADReplicationPartnerMetadata -Target (Get-ADDomainController -Filter *).Name |
Select-Object Server, Partner, LastReplicationSuccess
```

- **Get-ADFineGrainedPasswordPolicy** = retrieves Fine-Grained Password Policies (FGPP), which allow different password/lockout policies to apply to specific groups of users instead of a single domain-wide policy in `Get-ADDefaultDomainPasswordPolicy`.
    - **Get-ADDefaultDomainPasswordPolicy** = shows the single domain-wide default password policy (min length, complexity, lockout threshold), the AD equivalent of what Linux keeps in `/etc/security/pwquality.conf` and `/etc/login.defs`.
- **Get-ADObject** = the most generic AD retrieval cmdlet — can query any object type at all (not just users/computers/groups), useful for looking up objects by GUID, or for object classes without a dedicated cmdlet.

```
Get-ADObject -Filter "ObjectClass -eq 'organizationalUnit'"
```

- **dsacls / Active Directory ACL inspection** = while not a native PowerShell cmdlet, `dsacls.exe` (a legacy command-line tool) is still frequently invoked from PowerShell scripts to inspect or modify permissions directly on AD objects (as opposed to file/folder NTFS permissions, which use `icacls`/`Get-Acl` instead). For scripted AD ACL work, `Get-Acl`/`Set-Acl` also work against the `AD:` PSDrive path once the ActiveDirectory module is loaded.
- **Set-ADAccountControl** = modifies low-level userAccountControl flags on an account, giving access to specific behaviors that don't have a dedicated named parameter elsewhere (e.g., `-PasswordNeverExpires`, `-AccountNotDelegated`, `-TrustedForDelegation`).

```
Set-ADAccountControl -Identity svc_example -PasswordNeverExpires $true
```

- **Import-Module ActiveDirectory** = loads the AD cmdlets into a session (see the Scripting Fundamentals section for general module details). This module requires the RSAT: Active Directory Domain Services feature to be installed on the machine you're running it from (either a domain controller, or a management workstation with RSAT tools enabled).
- **New-ADServiceAccount / Install-ADServiceAccount** = creates and deploys a Group Managed Service Account (gMSA) — a special AD account type where Active Directory itself automatically manages and rotates the account's password on a schedule, eliminating the need to manually update a hardcoded service account password across every server that uses it (a major security improvement over traditional static service accounts).

```
New-ADServiceAccount -Name "gMSA-App01" -DNSHostName "gmsa-app01.contoso.com"
-PrincipalsAllowedToRetrieveManagedPassword "AppServers"
Install-ADServiceAccount -Identity "gMSA-App01"   # run on each server that needs to
use the gMSA
Test-ADServiceAccount -Identity "gMSA-App01"
```

    - **-PrincipalsAllowedToRetrieveManagedPassword** = specifies which computer accounts or security groups are permitted to retrieve the gMSA's current password — the account is otherwise completely unusable from any other machine.
    - **Install-ADServiceAccount** = must be run locally on **each** server that will use the gMSA, registering it with that machine before a service can actually run as that identity.
    - **Test-ADServiceAccount** = verifies the local machine can successfully retrieve and use the gMSA's current password, the standard first troubleshooting step if a service configured to run as a gMSA won't start.
- **Get-ADOptionalFeature / Enable-ADOptionalFeature (AD Recycle Bin)** = the AD Recycle Bin lets deleted AD objects be restored later with all their original attributes and group memberships intact, instead of relying on a full authoritative restore from backup. It's an optional feature that must be explicitly enabled once per forest (and, importantly, cannot be disabled again afterward).

```
Enable-ADOptionalFeature -Identity "Recycle Bin Feature" -Scope
ForestOrConfigurationSet -Target "contoso.com"
```

  - **Get-ADObject -Filter {isDeleted -eq $true} -IncludeDeletedObjects** = retrieves objects currently sitting in the Recycle Bin.
  - **Restore-ADObject** = restores a specific deleted object (found via the query above) back to an active state.
- **Get-ADReplicationSite / Get-ADReplicationSubnet** = retrieve Active Directory Sites and Services configuration — the site topology that governs replication scheduling between domain controllers in different physical locations (like separate remote sites) and helps clients locate their nearest domain controller.
  - **Get-ADReplicationSiteLink** = shows the configured replication links between sites, including their replication schedule and cost.
- **Get-ADTrust** = retrieves configured trust relationships between the current domain/forest and other domains or forests — relevant when troubleshooting authentication across a merger/acquisition scenario or a resource forest, or simply auditing which external trusts exist.
- **New-ADFineGrainedPasswordPolicy / Add-ADFineGrainedPasswordPolicySubject** = creates a Fine-Grained Password Policy (FGPP) with settings different from the domain-wide default, and applies it to a specific set of users or groups — for example, enforcing a stricter lockout threshold specifically for privileged admin accounts without changing the policy for every regular user in the domain.

```
New-ADFineGrainedPasswordPolicy -Name "AdminStrictPolicy" -Precedence 10
-MinPasswordLength 16 -LockoutThreshold 3
Add-ADFineGrainedPasswordPolicySubject -Identity "AdminStrictPolicy" -Subjects "Domain
Admins"
```

  - **-Precedence** = when a user is subject to multiple FGPPs, the one with the **lowest** precedence number wins — an easy detail to get backwards.
- **Get-ADRootDSE** = retrieves the Root DSE (DSA-Specific Entry) — a special LDAP entry exposing low-level metadata about the directory server itself, such as the current schema version, naming contexts, and supported LDAP controls. Rarely needed day-to-day, but useful when troubleshooting schema-version-related compatibility issues (e.g., confirming the schema version was actually updated after an AD-prep step during a domain functional level upgrade).

Microsoft Graph PowerShell SDK (Microsoft.Graph)

The Microsoft.Graph module is a large family of sub-modules (Users, Groups, Mail, Identity.SignIns, DeviceManagement, etc.) that wrap the Microsoft Graph REST API in native PowerShell cmdlets. This is the modern, actively-developed replacement for the older, now-deprecated AzureAD and MSOnline modules, and is the primary way to manage Entra ID (Azure AD), Intune, and Microsoft 365 resources from the command line today.

- **Connect-MgGraph** = authenticates the current PowerShell session to Microsoft Graph — every other Mg cmdlet requires this to run first.

```
Connect-MgGraph -Scopes "User.ReadWrite.All","Group.ReadWrite.All"
Connect-MgGraph -ClientId $AppId -TenantId $TenantId -CertificateThumbprint
$Thumbprint
```

  - **-Scopes** = an array of Graph API permission scopes being requested for this interactive sign-in session (ex. `User.Read.All`, `Group.ReadWrite.All`, `Directory.Read.All`). Interactive sign-in only grants permissions the signed-in admin's role already has and that have been consented to in the tenant.
  - **-ClientId / -TenantId / -CertificateThumbprint** = used for **unattended/app-only authentication** with a registered Entra ID App Registration and a certificate — required for any Graph script that needs to run unattended (e.g., a scheduled task), since interactive sign-in obviously can't prompt anyone.
  - **-ClientSecretCredential** = an alternative app-only auth method using a client secret instead of a certificate (certificates are the more secure, generally recommended option).

    - Scopes requested are **cumulative and cached** per session/token — if you get a permission error on a later cmdlet, you likely need to reconnect with additional scopes rather than assuming the module itself is broken.
- **Get-MgContext** = shows the current Graph connection's authentication context: which account/app is connected, which tenant, and exactly which scopes were granted for the current session — the first thing to check when a script that should have permission is throwing 'Insufficient privileges' errors.

```
Get-MgContext | Select-Object Account, TenantId, Scopes
```

- **Disconnect-MgGraph** = signs out of the current Graph session and clears the cached token — good practice at the end of a script, especially in shared/automation environments.
- **Find-MgGraphCommand** = searches for the correct Graph SDK cmdlet(s) that correspond to a specific REST API URI or a keyword — extremely useful given how large the Microsoft.Graph module family is, since a single REST endpoint sometimes maps to cmdlets spread across several sub-modules.

```
Find-MgGraphCommand -Command "Get-MgUser"
Find-MgGraphCommand -Uri "/users/{id}/manager"
```

- **Get-MgUser** = retrieves one or more Entra ID (Azure AD) user objects — the cloud/Graph equivalent of `Get-ADUser`, but querying Entra ID directly rather than on-premises AD.

```
Get-MgUser -UserId asample@contoso.com -Property DisplayName,JobTitle,Department
Get-MgUser -Filter "department eq 'IT'" -All
```

    - **-UserId** = accepts a user's UPN or Object ID to retrieve a single specific user.
    - **-Filter** = an OData filter string (note: different syntax from the AD module's PowerShell-style `-Filter`) used to search across multiple users by attribute value.
    - **-All** = returns all matching results across pagination automatically; without it, Graph cmdlets return only a default page size and you'd otherwise have to manually follow `@odata.nextLink` yourself.
    - **-Property** = Graph cmdlets return only a minimal default set of properties for performance reasons — you must explicitly request additional properties by name, similar in spirit to `-Properties` on `Get-ADUser`, but the two use different underlying default property sets.
- **New-MgUser / Update-MgUser / Remove-MgUser** = create, modify, or delete an Entra ID user account directly in the cloud.

```
$PasswordProfile = @{ Password = "Example#Pass1"; ForceChangePasswordNextSignIn =
$true }
New-MgUser -DisplayName "Alex Sample" -UserPrincipalName asample@contoso.com `
  -AccountEnabled -MailNickname asample -PasswordProfile $PasswordProfile
```

    - **-PasswordProfile** = a hashtable (not a SecureString like the AD module) specifying the initial password and whether the user must change it at next sign-in.
    - **Update-MgUser** uses the same `-Replace`-style pattern as `Set-ADUser` conceptually, but properties are passed as named parameters or a `-BodyParameter` hashtable matching the Graph API's JSON schema for a user object.
- **Get-MgGroup / New-MgGroup** = retrieves or creates Entra ID groups — including cloud-native Microsoft 365 Groups, which don't have a direct on-premises AD equivalent.
    - **-Filter "groupTypes/any(c:c eq 'Unified')"** = a common OData filter pattern used to find only Microsoft 365 (Unified) groups, as opposed to plain security groups.
    - **New-MgGroup -GroupTypes** = set to `@("Unified")` to create a Microsoft 365 Group, or left empty for a plain security group.
- **Get-MgGroupMember / New-MgGroupMember / Remove-MgGroupMember** = manage group membership in Entra ID.

```
Get-MgGroupMember -GroupId $GroupId -All
New-MgGroupMember -GroupId $GroupId -DirectoryObjectId $UserId
```

    - Unlike the AD module's `Add-ADGroupMember`, which accepts a friendly username, Graph cmdlets almost universally require the **Object ID (GUID)** of both the group and the member, not a display name or UPN — you'll frequently need to resolve a UPN to an Object ID with `Get-MgUser` first.
- **Get-MgUserMemberOf** = the reverse lookup of group membership — shows every group a specific user belongs to, the Graph equivalent of `Get-ADPrincipalGroupMembership`.
- **Get-MgDevice** = retrieves devices registered or joined to Entra ID (Hybrid Azure AD Join, Azure AD Join, or Azure AD Registered) — useful for auditing which devices are enrolled, their compliance state, and their trust type.
    - **-Filter "operatingSystem eq 'Windows'"** = a common filter example to narrow results by OS.
- **Get-MgUserLicenseDetail / Set-MgUserLicense** = manage Microsoft 365 license assignments for a user directly.

```
Get-MgUserLicenseDetail -UserId asample@contoso.com
Set-MgUserLicense -UserId asample@contoso.com -AddLicenses @{SkuId=$SkuId}
-RemoveLicenses @()
```

    - **Get-MgSubscribedSku** = lists the license SKUs (products) your tenant owns and how many seats are consumed vs. available — required to look up the exact SkuId GUID needed for `Set-MgUserLicense`.
- **Get-MgAuditLogSignIn / Get-MgAuditLogDirectoryAudit** = retrieves Entra ID sign-in logs and directory audit logs directly via Graph — useful for scripted security investigations (e.g., pulling all failed sign-ins for a specific user or IP range) without needing to manually export from the Entra admin portal.
    - Requires an Entra ID P1/P2 license in the tenant, since sign-in log retention and detail depend on licensing tier, same as when viewing them in the portal.
- **Get-MgDomain** = retrieves the custom domains configured/verified in the tenant, including their verification status and whether each is set as the default domain — a quick way to confirm exactly which domains (e.g., checking for something like a rogue subdomain) are registered against the tenant.
- **Get-MgOrganization** = retrieves tenant-level organization details: tenant ID, display name, verified domains, and technical/security contact information — the Graph equivalent of a quick tenant identity check.
- **Invoke-MgGraphRequest** = a generic escape hatch that lets you call **any** Graph API endpoint directly with a raw URI, even ones that don't yet have a dedicated cmdlet wrapper in the SDK — extremely useful for newly released Graph API features or beta endpoints.

```
Invoke-MgGraphRequest -Method GET -Uri
"https://graph.microsoft.com/v1.0/me/messages?\$top=5"
```

    - **-Method** = the HTTP verb (GET, POST, PATCH, DELETE).
    - **-Uri** = accepts either a full URL or a relative path against the current Graph API version.
    - **-Body** = a hashtable or JSON string for POST/PATCH request bodies.
- **Select-MgProfile (deprecated) / -ApiVersion** = in older SDK versions, `Select-MgProfile` was used to switch between the `v1.0` and `beta` Graph API endpoints globally for the session. In current SDK versions this has been removed in favor of specifying `-ApiVersion beta` directly on individual cmdlets that support it, or using the beta endpoint explicitly with `Invoke-MgGraphRequest`.
- **Get-MgDeviceManagementManagedDevice** = retrieves Intune-managed device records (Microsoft.Graph.DeviceManagement sub-module) — including compliance state, enrollment type, last check-in, and OS version, the Graph-native way to script device fleet reporting instead of exporting manually from the Intune admin center.

```
Get-MgDeviceManagementManagedDevice -Filter "complianceState eq 'noncompliant'"
```

    - **Sync-MgDeviceManagementManagedDevice** = forces an immediate policy/compliance sync on a specific device instead of waiting for its normal check-in interval.
    - **Remove-MgDeviceManagementManagedDevice** = retires/wipes a device record from Intune management.

- **Get-MgApplication / New-MgApplication** = manage Entra ID App Registrations — the identity objects representing custom applications, scripts, and service integrations that authenticate against the tenant (including the app registration typically created to support certificate-based unattended `Connect-MgGraph`/`Connect-ExchangeOnline` authentication).
    - **Get-MgServicePrincipal** = the related, but distinct, object representing an app registration's actual presence/permissions *within a specific tenant* (an App Registration is the global definition; a Service Principal is its local instantiation in your tenant).
    - **New-MgApplicationPassword / New-MgApplicationKey** = generates a new client secret or uploads a new certificate credential for an app registration, used when rotating credentials for a scheduled automation script.
- **Get-MgIdentityConditionalAccessPolicy** = retrieves configured Conditional Access policies, letting you script an inventory/audit of exactly which sign-in conditions (location, device compliance, risk level) trigger MFA requirements or block access entirely — useful for periodically validating that CA policies haven't silently drifted from an approved baseline.
- **Get-MgDirectoryRole / Add-MgDirectoryRoleMember** = manage Entra ID built-in administrative directory roles (Global Administrator, User Administrator, Helpdesk Administrator, etc.) — the cloud equivalent of assigning someone to a privileged AD security group, but governing tenant-wide administrative permissions instead of on-premises resource access.
    - **Get-MgDirectoryRoleMember** = lists who currently holds a specific administrative role — a standard periodic security audit query to confirm privileged role membership hasn't grown beyond what's expected.
- **Get-MgUserAuthenticationMethod** = retrieves the registered authentication methods (Authenticator app, phone, FIDO2 security key, Windows Hello for Business) for a specific user — useful for helpdesk troubleshooting of MFA-related sign-in issues, or auditing which users still only have a weaker authentication method registered.
- **Revoke-MgUserSignInSession** = immediately invalidates all of a user's currently active refresh tokens across every device and application, forcing them to fully re-authenticate everywhere on their next access attempt — a critical incident-response action when an account is suspected of compromise, since simply resetting the password alone does not invalidate already-issued tokens.

```
Revoke-MgUserSignInSession -UserId asample@contoso.com
```

- **Get-MgGroupOwner / New-MgGroupOwner** = manage group **ownership** (as distinct from group **membership**) — owners of a Microsoft 365 Group can manage the group's membership and settings themselves without needing separate directory-wide admin permissions, a common self-service delegation pattern.
- **Batching, Throttling & Pagination** = a few Graph-specific behavioral concepts worth understanding conceptually, since they aren't single cmdlets but affect how you should write reliable Graph scripts.
    - **Pagination** = Graph API results are paged by default (a maximum page size per request); the `-All` switch on most `Get-Mg*` cmdlets handles following the pagination links (`@odata.nextLink`) automatically so you don't have to write that logic yourself.
    - **Throttling** = Graph enforces per-tenant and per-app rate limits; a throttled request returns an HTTP 429 status with a `Retry-After` header telling you how long to wait — production scripts making heavy Graph calls should implement retry logic that respects this header rather than hammering the API repeatedly.
    - **$select and -Property** = always request only the specific properties you actually need rather than pulling full objects by default — this significantly reduces both response payload size and the chance of hitting throttling limits on large bulk queries.

Azure PowerShell (Az Module)

The Az module is the modern, actively-maintained set of cmdlets for managing Azure resources from PowerShell, replacing the older, now-deprecated AzureRM module. It's organized into sub-modules by service (Az.Compute, Az.Storage, Az.Network, Az.KeyVault, etc.), all installed together as the umbrella 'Az' module.

- **Connect-AzAccount** = authenticates the current session to Azure — the first command in virtually every Az script.

```
Connect-AzAccount
Connect-AzAccount -Identity   # used from inside an Azure resource with a Managed
Identity
Connect-AzAccount -ServicePrincipal -Credential $Cred -Tenant $TenantId
```

  - **-Identity** = authenticates using an Azure **Managed Identity** instead of a user or app secret — the recommended, credential-free way to authenticate a script running *from inside* Azure (e.g., an Azure Automation runbook or a VM).
  - **-ServicePrincipal** = authenticates as an app registration/service principal using a client secret or certificate, required for unattended scripts running **outside** Azure.
  - **-TenantId / -SubscriptionId** = explicitly targets a specific tenant/subscription at sign-in time, useful for accounts with access to multiple tenants.
- **Get-AzContext / Set-AzContext** = shows or changes the currently active subscription/tenant context for the session — the Azure equivalent of `Get-MgContext`, and essential to check in any environment where a single account may have access to multiple subscriptions.

```
Get-AzContext
Set-AzContext -Subscription "Production"
```

- **Get-AzSubscription** = lists all Azure subscriptions the currently signed-in account has access to — the natural first step before `Set-AzContext` when you're not sure of the exact subscription name/ID.
- **Get-AzResourceGroup / New-AzResourceGroup / Remove-AzResourceGroup** = manage Resource Groups — the fundamental organizational container in Azure that almost every other resource must belong to.

```
New-AzResourceGroup -Name "RG-Example" -Location "eastus"
```

  - **Remove-AzResourceGroup** = deletes the resource group **and every resource inside it** — one of the most dangerous single commands in Azure PowerShell, since it cascades to everything the group contains.
- **Get-AzResource** = a generic cmdlet that retrieves **any** Azure resource across all resource types — useful for broad inventory/auditing queries that span multiple resource types at once (VMs, storage accounts, NICs, etc.) rather than needing a separate cmdlet call per resource type.
  - **-ResourceGroupName** = restricts results to resources within a specific resource group.
  - **-ResourceType** = filters to a specific resource type (ex. `Microsoft.Compute/virtualMachines`).
- **Get-AzVM / New-AzVM / Start-AzVM / Stop-AzVM / Restart-AzVM** = manage the lifecycle of Azure Virtual Machines.
  - **Get-AzVM -Status** = includes the current power state (running/deallocated) in the output, which is not included by default.
  - **New-AzVM** = provisions a new VM; in practice, complex VM deployments are almost always built up from several supporting objects first (a `New-AzVMConfig`, NIC, OS disk, and network settings combined together) rather than one single flat cmdlet call.
  - **Stop-AzVM -Force** = stops and **deallocates** the VM (releasing its compute resources and stopping compute billing) — different from just shutting down the guest OS, which would leave the VM allocated and still billing for compute.
  - **Restart-AzVM** = restarts a running VM without deallocating it.
- **Get-AzStorageAccount / New-AzStorageAccount** = manage Azure Storage Accounts, the container for blob, file, table, and queue storage services.
  - **Get-AzStorageContainer / Get-AzStorageBlob** = (from Az.Storage) drill down further into a specific storage account's blob containers and individual blobs once you've retrieved the storage account context with `New-AzStorageContext` or `Get-AzStorageAccount`.
- **Get-AzKeyVault / Get-AzKeyVaultSecret / Set-AzKeyVaultSecret** = manage Azure Key Vault, used to securely store secrets, certificates, and keys rather than embedding them in plain text inside scripts.

```
$Secret = Get-AzKeyVaultSecret -VaultName "KV-Example" -Name "SvcAccountPassword"
-AsPlainText
Set-AzKeyVaultSecret -VaultName "KV-Example" -Name "NewApiKey" -SecretValue
(ConvertTo-SecureString "example-secret" -AsPlainText -Force)
```

    - **-AsPlainText** = returns the secret's raw string value directly, instead of the default `SecureString` wrapper object — required when you actually need to use the value, not just check that it exists.
- **Get-AzADUser / Get-AzADGroup / Get-AzADServicePrincipal** = a thinner, Azure-Resource-Manager-scoped set of Entra ID lookups bundled inside the Az module, primarily used for assigning Azure RBAC roles to users/groups/service principals — for full-featured Entra ID user/group management, the Microsoft.Graph module is the primary, more complete tool.
- **Get-AzRoleAssignment / New-AzRoleAssignment / Remove-AzRoleAssignment** = manage Azure RBAC (Role-Based Access Control) — assigning built-in or custom roles (Owner, Contributor, Reader, or more granular roles) to a user, group, or service principal at a specific scope (subscription, resource group, or individual resource).

```
New-AzRoleAssignment -SignInName asample@contoso.com -RoleDefinitionName "Reader"
-ResourceGroupName "RG-Example"
```

    - **-Scope** = an alternative to `-ResourceGroupName` that accepts the full Azure Resource Manager scope path directly, letting you assign a role at the subscription, resource group, or individual resource level.
- **Get-AzVirtualNetwork / Get-AzNetworkSecurityGroup** = retrieve Azure Virtual Network (VNet) and Network Security Group (NSG) configurations — the Azure equivalent of inspecting network topology and firewall rules, conceptually similar to combining `ip addr`/routes with `firewall-cmd --list-all` on-prem, but for cloud-native networking constructs.
- **Test-AzNetworkWatcherConnectivity / Get-AzNetworkWatcher** = Azure's built-in network diagnostics service (Network Watcher), letting you test connectivity between two Azure resources, capture packets, or check effective NSG rules on a NIC directly from PowerShell — roughly the cloud-native equivalent of running `tcpdump` or `Test-NetConnection` against something you can't directly RDP/SSH into.
- **Get-AzADAppRegistration / New-AzADApplication** = manage Entra ID App Registrations from within the Az module context — commonly used when setting up a service principal for unattended automation (e.g., the app registration and certificate needed for `Connect-MgGraph -CertificateThumbprint` in a scheduled script).
- **Get-AzActivityLog** = retrieves the Azure Activity Log — a record of subscription-level control-plane operations (who created/modified/deleted which resource and when), the Azure equivalent of an audit trail, useful for investigating unexpected changes to production resources.
    - **-ResourceGroupName** = scopes the log query to a specific resource group.
    - **-StartTime / -EndTime** = bounds the time range of the query.
- **Export-AzResourceGroup / New-AzResourceGroupDeployment** = manage Azure Resource Manager (ARM) templates — exporting a resource group's current configuration as a reusable JSON template, or deploying a new/updated set of resources from a template file. This is Azure's native Infrastructure-as-Code mechanism, conceptually parallel to the OpenTofu workflow (`init`/`plan`/`apply`) but using ARM/Bicep templates specifically rather than HCL.

```
New-AzResourceGroupDeployment -ResourceGroupName "RG-Example" -TemplateFile
".\main.json" -TemplateParameterFile ".\params.json"
```

- **Update-AzConfig** = configures Az module-wide behavior defaults for the current user, such as whether to display upcoming breaking-change warnings, or whether to automatically survey installed modules for updates — a housekeeping cmdlet worth knowing exists when Az's verbose warning banners get in the way of clean script output.
- **Get-AzVMSize / Get-AzVMExtension** = inspect VM sizing options and installed VM extensions.
    - **Get-AzVMSize -Location \<region\>** = lists all available VM size SKUs (vCPU/RAM combinations) for a specific Azure region, since availability varies by region and not every size is offered everywhere.

    - **Get-AzVMExtension** = lists extensions installed on a VM (e.g., the Azure Monitor Agent, a custom script extension, or antimalware extension) — VM extensions are small agents that run additional configuration/monitoring tasks inside the guest OS.
- **Get-AzPublicIpAddress / Get-AzLoadBalancer** = retrieve Azure networking resources related to inbound connectivity — public IP address objects and load balancer configurations respectively, used when auditing what's actually internet-facing in a subscription.
- **New-AzVirtualNetwork / New-AzVirtualNetworkSubnetConfig** = provision a new Azure Virtual Network and its subnets — typically built together as a multi-step pipeline (define subnet configs first, then pass them into the VNet creation call), similar in spirit to defining VLANs before assigning them to switch ports.

```
$Subnet = New-AzVirtualNetworkSubnetConfig -Name "Subnet-App" -AddressPrefix
"198.18.1.0/24"
New-AzVirtualNetwork -Name "VNet-Example" -ResourceGroupName "RG-Example" -Location
"eastus" -AddressPrefix "198.18.0.0/16" -Subnet $Subnet
```

- **Get-AzMetric / New-AzMetricAlertRuleV2** = query historical resource performance metrics (CPU, disk IOPS, network throughput) directly from Azure Monitor, or create an alert rule that fires when a metric crosses a defined threshold — the Azure-native equivalent in purpose to configuring a threshold alert against `mpstat`/`iostat` data collected locally on a Linux box, but fully managed by the platform.

```
Get-AzMetric -ResourceId $VMResourceId -MetricName "Percentage CPU" -TimeGrain
00:05:00
```

- **Get-AzAutomationAccount / Start-AzAutomationRunbook** = Azure Automation is a cloud-hosted service for running PowerShell/Python runbooks on a schedule or on-demand without needing a dedicated always-on VM — commonly used for exactly the kind of unattended, certificate-authenticated scripts described in the Graph and Exchange Online sections above (scheduled compliance reports, cleanup jobs, off-hours automation).
    - **Start-AzAutomationRunbook** = manually triggers a runbook to run immediately, outside its normal schedule — useful for testing a runbook before finalizing its scheduled trigger.
- **New-AzStorageContext / New-AzStorageAccountSASToken** = **New-AzStorageContext** creates the authentication context object required by most `Az.Storage` cmdlets to actually connect to a specific storage account's data plane (as opposed to just its resource-manager configuration). **New-AzStorageAccountSASToken** generates a Shared Access Signature — a time-limited, scope-limited token that can be handed to a third party (or embedded in a script) to grant temporary access to specific storage resources without exposing the storage account's full access keys.
- **Get-AzSqlServer / Get-AzSqlDatabase** = retrieve Azure SQL Database server and database resources — used for basic inventory/auditing of Azure-hosted SQL workloads, including firewall rule configuration (`Get-AzSqlServerFirewallRule`) controlling which IP ranges can reach the database.

Exchange Online Management Module (ExchangeOnlineManagement)

The ExchangeOnlineManagement module (commonly abbreviated EXO) provides the PowerShell cmdlets for administering Exchange Online mailboxes, mail flow, and compliance settings in Microsoft 365 — the cloud successor to the on-premises Exchange Management Shell. Most cmdlets here retain the same naming conventions long-time Exchange admins already know from on-prem.

- **Connect-ExchangeOnline** = authenticates the session to Exchange Online — required before any other EXO cmdlet will function.

```
Connect-ExchangeOnline -UserPrincipalName admin@contoso.com
Connect-ExchangeOnline -CertificateThumbprint $Thumbprint -AppId $AppId -Organization
contoso.onmicrosoft.com
```

    - **-UserPrincipalName** = interactive sign-in as a specific admin account (will prompt for MFA if required by conditional access).

    - **-CertificateThumbprint / -AppId / -Organization** = certificate-based app-only authentication for unattended scripts (e.g., a scheduled compliance report), avoiding the need for a stored username/password.
    - **-ShowBanner:$false** = suppresses the informational banner shown on connect, useful to keep automated script output/logs clean.
    - **Disconnect-ExchangeOnline** = ends the session and clears the connection — good practice at the end of any script, particularly important in shared automation environments.
- **Get-Mailbox** = retrieves one or more Exchange Online mailbox objects — the single most frequently used cmdlet in the module.

```
Get-Mailbox -Identity asample@contoso.com
Get-Mailbox -RecipientTypeDetails SharedMailbox -ResultSize Unlimited
```

    - **-Identity** = accepts an email address, alias, display name, or GUID to retrieve a single mailbox.
    - **-RecipientTypeDetails** = filters by mailbox type (UserMailbox, SharedMailbox, RoomMailbox, EquipmentMailbox, DiscoveryMailbox).
    - **-ResultSize Unlimited** = required to return more than the default page size (1,000) of results — a very common gotcha where a script silently appears to be 'missing' mailboxes simply because this wasn't specified.
    - **-Anr** = ambiguous name resolution; searches across multiple name-related fields at once when you're not sure of the exact identity string.
- **Set-Mailbox** = modifies mailbox settings and properties — a very broad cmdlet covering everything from forwarding rules to size quotas to litigation hold.

```
Set-Mailbox -Identity asample@contoso.com -ForwardingSmtpAddress
external.user@example.net -DeliverToMailboxAndForward $true
Set-Mailbox -Identity asample@contoso.com -LitigationHoldEnabled $true
```

    - **-ForwardingSmtpAddress** vs **-ForwardingAddress** = the SMTP variant forwards to any external or internal address by raw SMTP address; the non-SMTP variant requires the target to be a resolvable recipient object within the org.
    - **-DeliverToMailboxAndForward** = if `$true`, keeps a copy in the original mailbox in addition to forwarding; if `$false`, mail is forwarded only and not retained locally.
    - **-LitigationHoldEnabled** = places the mailbox on hold, preserving all item versions (including deleted/edited items) for legal/compliance purposes, independent of the user's own retention actions.
    - **-Type** = converts a mailbox type (ex. converting a former employee's mailbox to `Shared` so it stops consuming a paid license while remaining accessible to a team).
- **New-Mailbox / Remove-Mailbox / New-RemoteMailbox** = create or remove Exchange Online mailboxes.
    - **New-Mailbox** = creates a new cloud-only mailbox and associated Entra ID user account together in one step.
    - **New-RemoteMailbox** = used specifically in **hybrid** environments — it creates the on-premises AD user object with the correct Exchange attributes so it syncs up and provisions a matching Exchange Online mailbox, rather than creating the mailbox directly in the cloud.
    - **Remove-Mailbox** = deletes a mailbox; for a licensed user this typically also requires separately removing the underlying Entra ID user account or converting the mailbox to shared first, depending on the intended outcome.
- **Get-MailboxPermission / Add-MailboxPermission / Remove-MailboxPermission** = manage Full Access delegate permissions on a mailbox — who else can open and act inside someone else's mailbox (e.g., an assistant with full access to an executive's mailbox, or a team with access to a shared mailbox).

```
Add-MailboxPermission -Identity "Shared Mailbox Example" -User bexample@contoso.com
-AccessRights FullAccess -AutoMapping $true
```

    - **-AutoMapping** = controls whether the mailbox automatically appears in the granted user's Outlook profile without them needing to manually add it — setting this to `$false` is common when granting broad access that shouldn't visually clutter everyone's Outlook (e.g., programmatic/bulk access grants).
- **Add-RecipientPermission** = grants **Send As** permission on a mailbox — different from Full Access; this only allows sending mail that appears to come from the mailbox, without granting the ability to actually open and read its contents.

```
Add-RecipientPermission -Identity "Shared Mailbox Example" -Trustee
bexample@contoso.com -AccessRights SendAs
```

- **Get-MailboxPermission vs Send on Behalf** = worth noting as a conceptual trio since these three permission types are frequently confused: **Full Access** (open/read the whole mailbox), **Send As** (send mail that looks like it came directly from the mailbox, no 'on behalf of' indicator), and **Send on Behalf** (managed via `Set-Mailbox -GrantSendOnBehalfTo`, which sends mail clearly marked as 'X on behalf of Y' in the recipient's client).
- **Get-MailboxStatistics** = returns mailbox size, item count, and last logon time — used constantly for storage/quota auditing and for confirming a mailbox is actually being used before decommissioning it.

```
Get-MailboxStatistics -Identity asample@contoso.com | Select-Object DisplayName,
TotalItemSize, ItemCount, LastLogonTime
```

- **Get-MessageTrace / Get-MessageTraceDetail** = traces the delivery path and status of email messages sent through Exchange Online in the last 10 days by default (or up to 90 days via the newer historical search cmdlets) — the primary tool for answering 'did this email actually get delivered, and if not, why.'

```
Get-MessageTrace -SenderAddress noreply@example.net -StartDate (Get-Date).AddDays(-2)
-EndDate (Get-Date)
Get-MessageTraceDetail -MessageTraceId $Id -RecipientAddress user@contoso.com
```

    - **Get-MessageTraceDetail** = drills into a single specific message (identified by its MessageTraceId) to show the detailed step-by-step delivery events, including any rejection reasons.
- **Get-DistributionGroup / New-DistributionGroup** = manage classic Distribution Groups (as opposed to Microsoft 365 Groups, which are managed instead through the Microsoft.Graph or Teams modules).

```
New-DistributionGroup -Name "Contoso-Announcements" -Members
asample@contoso.com,cdemo@contoso.com
```

    - **Get-DistributionGroupMember / Add-DistributionGroupMember / Remove-DistributionGroupMember** = manage a distribution group's membership list.
- **Get-TransportRule / New-TransportRule** = manage Exchange Online mail flow rules (transport rules) — the cloud-side rules engine that inspects, modifies, or blocks messages in transit based on conditions (sender, recipient, subject/body keywords, attachment type), heavily used for things like DLP-style blocking or automatic disclaimers.

```
Get-TransportRule | Select-Object Name, State, Priority
```

- **Get-InboundConnector / Get-OutboundConnector / New-InboundConnector / New-OutboundConnector** = manage mail flow connectors — used heavily for hybrid mail flow and for partner/vendor-specific SMTP relationships that require special routing or TLS enforcement outside the standard internet mail flow path.
    - **Inbound connectors** = configure how mail is *accepted* from a specific external partner (e.g., enforcing that mail claiming to be from a specific vendor must arrive from their specific IP range or with a specific certificate).
    - **Outbound connectors** = configure how mail is *sent* to a specific external partner (e.g., forcing TLS, or routing through a specific smart host for a partner requiring it).

- **Get-AcceptedDomain** = lists every domain the tenant is configured to accept mail for, along with its type (Authoritative, InternalRelay, ExternalRelay) — the first thing to check when investigating whether a specific domain/subdomain is properly configured to receive or relay mail (e.g., confirming whether a subdomain was ever added as an accepted domain in the first place).
- **Get-DkimSigningConfig / Set-DkimSigningConfig** = manages DKIM signing configuration per domain in Exchange Online — used to check whether DKIM is enabled for a domain and to view/rotate the CNAME selector records that must be published in DNS.

```
Get-DkimSigningConfig -Identity contoso.com | Select-Object Domain, Enabled,
Selector1CNAME, Selector2CNAME
```

    - **-Enabled** = toggles whether Exchange Online actually signs outbound mail for that domain with DKIM (the CNAME records must already be correctly published in DNS before enabling this, or mail signing will fail silently).
- **Get-HostedContentFilterPolicy / Get-HostedOutboundSpamFilterPolicy** = retrieve Exchange Online Protection (EOP) anti-spam policy configuration — the cloud spam filter rules governing what's marked as spam/bulk/phishing and the resulting actions (quarantine, junk folder, delete).
- **Get-AntiPhishPolicy / Get-SafeAttachmentPolicy / Get-SafeLinksPolicy** = retrieve Microsoft Defender for Office 365 policy configurations (impersonation protection, sandboxed attachment detonation, and link rewriting/scanning respectively) — these require a Defender for Office 365 (Plan 1 or 2) license to actually be enforced, distinct from the baseline anti-spam policies available on every tenant.
- **New-ComplianceSearch / Start-ComplianceSearch / New-ComplianceSearchAction** = technically part of the related Security & Compliance PowerShell module (connected via `Connect-IPPSSession` rather than `Connect-ExchangeOnline`), but used constantly alongside EXO for eDiscovery — searching across mailboxes for specific content (by keyword, sender, date range) and then taking an action on the results, such as exporting or purging matching items.

```
Connect-IPPSSession -UserPrincipalName admin@contoso.com
New-ComplianceSearch -Name "Compliance-Search-Example" -ExchangeLocation All
-ContentMatchQuery 'subject:"example keyword"'
Start-ComplianceSearch -Identity "Compliance-Search-Example"
```

- **Set-CASMailbox** = configures Client Access Settings for a mailbox — protocol-level access controls like whether ActiveSync, OWA, IMAP, POP, or MAPI are permitted for a specific user, commonly used as a security hardening step to disable legacy authentication protocols (IMAP/POP) on a per-mailbox basis.

```
Set-CASMailbox -Identity asample@contoso.com -ImapEnabled $false -PopEnabled $false
```

- **Get-EXOMailbox / Get-EXORecipient / Get-EXOMailboxPermission** = a set of newer, higher-performance 'EXO'-prefixed cmdlets introduced specifically to be faster and more scalable than their classic counterparts (`Get-Mailbox`, `Get-Recipient`, `Get-MailboxPermission`) when querying against very large tenants — functionally similar output, but recommended for bulk/reporting scripts over thousands of mailboxes where performance matters.
- **Get-QuarantineMessage / Release-QuarantineMessage** = manage messages held in the Exchange Online Protection quarantine (flagged as spam, phishing, malware, or blocked by a transport/DLP rule) — one of the most common day-to-day admin tasks, releasing legitimate messages that were incorrectly quarantined.

```
Get-QuarantineMessage -StartReceivedDate (Get-Date).AddDays(-2) -RecipientAddress
user@contoso.com
Release-QuarantineMessage -Identity $MessageId -ReleaseToAll
```

    - **-ReleaseToAll** = releases the message to every original intended recipient at once, rather than only the specific recipient identity queried.
- **Get-DlpPolicy / New-DlpPolicy / Get-DlpComplianceRule** = manage Data Loss Prevention (DLP) policies (technically part of the Security & Compliance PowerShell module via `Connect-IPPSSession`, but used constantly alongside Exchange Online mail flow work) — the rules engine that detects sensitive data patterns (credit card

numbers, SSNs, specific keywords) in transit or at rest and can block, encrypt, or flag matching content, directly relevant to any project involving exceptions or exclusions from standard DLP enforcement for a specific shared mailbox workflow.

```
Get-DlpPolicy | Select-Object Name, Mode, Enabled
Get-DlpComplianceRule -Policy "Example Data Protection"
```

    - **-Mode** = a policy can be in `Enable` (actively enforcing), `TestWithNotifications` (simulating and alerting without blocking), or `TestWithoutNotifications` (fully silent simulation) — critical to check before assuming a policy is actually blocking anything in production.
- **Get-RetentionPolicy / New-RetentionPolicyTag** = manage Messaging Records Management (MRM) retention policies — rules governing how long mail items are retained before being automatically archived or permanently deleted, distinct from (but often used alongside) Litigation Hold and Microsoft Purview retention labels/policies.
    - **New-RetentionPolicyTag** = defines an individual retention action/duration (e.g., 'delete items in Deleted Items after 30 days'), which is then bundled into a `RetentionPolicy` and applied to mailboxes.
- **Get-Label / Get-LabelPolicy** = (Security & Compliance module) retrieve Microsoft Purview Information Protection sensitivity labels and their publishing policies — the labels end users see and apply in Outlook/Office apps (e.g., 'Confidential', 'Internal Use Only') that can automatically trigger encryption or access restrictions on labeled content.
- **Get-MobileDevice / Get-MobileDeviceStatistics** = retrieve mobile devices connected to a mailbox via Exchange ActiveSync (EAS), along with sync status and last successful sync time — useful for auditing which devices are actively syncing corporate mail, or troubleshooting a user's phone that's stopped receiving new mail.
    - **Remove-MobileDevice** = removes a specific device partnership, forcing it to be re-provisioned (and re-accept any device access policy) before it can sync again — a common step when decommissioning a lost or replaced phone.
- **Get-MailboxAutoReplyConfiguration / Set-MailboxAutoReplyConfiguration** = view or configure a mailbox's Automatic Replies (Out of Office) settings directly from PowerShell — useful for bulk-setting an out-of-office message across multiple mailboxes (e.g., a planned-closure notice) without needing each user to configure it individually in Outlook.

```
Set-MailboxAutoReplyConfiguration -Identity asample@contoso.com -AutoReplyState
Enabled -InternalMessage "I am out of the office."
```

- **Get-RemoteDomain / Set-RemoteDomain** = manage remote domain configuration objects controlling mail flow behavior specifically for messages sent to a particular external domain (e.g., disabling automatic-reply/out-of-office messages from being sent to a specific partner domain, or enforcing TLS for messages to that domain).
- **Get-OrganizationConfig / Set-OrganizationConfig** = view or modify tenant-wide Exchange Online organization settings — a broad cmdlet covering dozens of org-level toggles, from default mailbox regional settings to feature-level enablement flags (such as Direct Send behavior, focused inbox defaults, or read receipt handling), the kind of setting checked when investigating tenant-wide mail flow behavior that isn't specific to any single mailbox or connector.
- **Test-Mflow / Test-SmtpConnectivity** = built-in Exchange Online connectivity diagnostic cmdlets used to send test messages through the mail flow pipeline and validate connectivity/delivery end-to-end, useful as a scripted health check independent of manually sending a test email from Outlook.
- **Get-EXOMailboxFolderPermission / Add-MailboxFolderPermission** = manage granular, folder-level permissions within a mailbox (as opposed to whole-mailbox Full Access) — most commonly used to grant Calendar or specific subfolder access (e.g., `Reviewer`, `Editor`, `PublishingEditor` access levels) without exposing the entire mailbox.

```
Add-MailboxFolderPermission -Identity "asample:\Calendar" -User "cdemo@contoso.com"
-AccessRights Reviewer
```

- **Get-ConnectionInformation** = shows details about the current active Exchange Online / Security & Compliance PowerShell connection(s) in the session — useful in scripts that need to confirm a connection is still valid/active

before proceeding, or when troubleshooting a stale session left open from an earlier `Connect-ExchangeOnline` call.

Additional Sysadmin Modules & Tools

Beyond AD, Graph, Azure, and Exchange Online, these are the other built-in and commonly-installed PowerShell modules a Windows sysadmin runs into constantly — native storage/disk management, firewall rules, certificates, endpoint security, DNS/DHCP server administration, printing, and general PowerShell environment tooling.

- **Get-Disk / Get-Partition / Get-Volume** = native PowerShell storage cmdlets (Storage module) for inspecting physical disks, partitions, and volumes — the structured-object equivalent of `diskpart`'s `list disk`/`list volume`/`list partition` commands.
    - **Get-Disk** = lists physical/virtual disks, their size, partition style (GPT/MBR), and health status.
    - **Get-Partition** = lists partitions on a disk, including drive letter assignment and size.
    - **Get-Volume** = lists volumes with filesystem type, health status, and free/used space — the modern replacement for parsing `Get-CimInstance Win32_LogicalDisk` for basic space checks.
- **Initialize-Disk / New-Partition / Format-Volume** = the modern PowerShell-native disk provisioning pipeline, replacing the interactive `diskpart` workflow with fully scriptable cmdlets.

```
Initialize-Disk -Number 2 -PartitionStyle GPT
New-Partition -DiskNumber 2 -UseMaximumSize -AssignDriveLetter
Format-Volume -DriveLetter F -FileSystem NTFS -NewFileSystemLabel "Data"
```

    - **Resize-Partition** = grows or shrinks an existing partition — the PowerShell-native equivalent of `diskpart`'s `extend` command.
- **Get-NetFirewallRule / New-NetFirewallRule / Set-NetFirewallRule** = native PowerShell cmdlets (NetSecurity module) for querying and managing Windows Defender Firewall rules — the structured-object alternative to `netsh advfirewall`.

```
New-NetFirewallRule -DisplayName "Allow LDAPS" -Direction Inbound -Protocol TCP
-LocalPort 636 -Action Allow
```

    - **-Direction** = Inbound or Outbound.
    - **-Action** = Allow or Block.
    - **-Profile** = restricts the rule to specific network profiles (Domain, Private, Public).
    - **Get-NetFirewallRule | Where-Object { $_.Enabled -eq 'True' -and $_.Direction -eq 'Inbound' }** = a common audit query for reviewing all active inbound rules.
- **Get-BitLockerVolume / Enable-BitLocker / Suspend-BitLocker** = manage BitLocker full-disk encryption from PowerShell — checking encryption status, enabling protection on a volume, and temporarily suspending it (e.g., during a firmware update that would otherwise trigger recovery-key prompt on next boot).
    - **Get-BitLockerVolume** = shows encryption status, percentage encrypted, and protector types for each volume.
    - **Enable-BitLocker -MountPoint C: -RecoveryPasswordProtector** = turns on encryption for a volume with a randomly generated recovery password protector.
    - **Backup-BitLockerKeyProtector** = backs up a volume's recovery key to Active Directory or Entra ID, ensuring it's centrally recoverable if the end user loses it.
- **New-SelfSignedCertificate / Get-ChildItem Cert:** = PowerShell's native certificate management tools, operating through the special `Cert:` PSDrive provider rather than a traditional filesystem path.

```
New-SelfSignedCertificate -DnsName "testserver.contoso.com" -CertStoreLocation
"Cert:\LocalMachine\My"
Get-ChildItem -Path Cert:\LocalMachine\My | Where-Object { $_.NotAfter -lt
(Get-Date).AddDays(30) }
```

    - **New-SelfSignedCertificate** = generates a self-signed certificate, commonly used for internal testing or lab environments (not for production public-facing services, where a CA-issued cert is required).
    - **Cert:\LocalMachine\My** = the computer's personal certificate store; **Cert:\CurrentUser\My** = the current logged-in user's personal store.
    - The `Get-ChildItem` example above is the classic 'find certificates expiring soon' script pattern — checking `.NotAfter` against a threshold date, directly analogous to the Linux workflow of parsing `openssl x509 -enddate` output.
- **Get-MpComputerStatus / Start-MpScan / Update-MpSignature** = manage Microsoft Defender Antivirus directly from PowerShell (Defender module) — the built-in Windows equivalent in purpose to Linux's ClamAV command set.
    - **Get-MpComputerStatus** = shows whether real-time protection is enabled, signature version/age, and last scan time.
    - **Start-MpScan -ScanType QuickScan|FullScan** = triggers an on-demand scan.
    - **Update-MpSignature** = manually forces a signature definition update (the Defender equivalent of Linux's `freshclam`).
    - **Add-MpPreference -ExclusionPath** = adds a folder/file exclusion — commonly needed for server roles/applications known to conflict with real-time scanning (e.g., certain database or backup software directories).
- **Compress-Archive / Expand-Archive** = native PowerShell cmdlets for creating and extracting .zip archives — the built-in equivalent of Linux's `zip`/`unzip` (or `tar`), without needing any external compression utility installed.

```
Compress-Archive -Path "C:\Logs\*.log" -DestinationPath "C:\Archive\Logs_$(Get-Date
-Format yyyyMMdd).zip"
Expand-Archive -Path "C:\Archive\Logs.zip" -DestinationPath "C:\Restored"
```

    - Only supports the .zip format natively — for other archive formats (7z, tar.gz), you'd still need a third-party module or call out to an external tool like `7z.exe`.
- **DnsServer module (Get-DnsServerZone / Get-DnsServerResourceRecord)** = manage Windows DNS Server roles directly from PowerShell — used constantly for auditing DNS zones and records on a domain controller running the DNS Server role.

```
Get-DnsServerZone
Get-DnsServerResourceRecord -ZoneName "contoso.com" -RRType A
```

    - **Add-DnsServerResourceRecordA / Add-DnsServerResourceRecordCName** = create A or CNAME records directly, useful for scripting DNS record creation for new subdomains (e.g., setting up a `bounces` subdomain's CNAME for a mail vendor) without touching the DNS Manager GUI.
    - **Get-DnsServerForwarder** = shows configured DNS forwarders (the upstream/external DNS servers this DNS server queries for names it doesn't host itself).
- **DhcpServer module (Get-DhcpServerv4Scope / Get-DhcpServerv4Lease)** = manage the Windows DHCP Server role from PowerShell — inspecting scopes, lease utilization, and reservations across multiple subnets.
    - **Get-DhcpServerv4Scope** = lists configured DHCP scopes and their address ranges.
    - **Get-DhcpServerv4ScopeStatistics** = shows how close a scope is to running out of available addresses — the key metric for catching subnet exhaustion before it causes outages at a remote site.
    - **Get-DhcpServerv4Lease** = lists current active leases within a scope, including client hostname and MAC address.
    - **Add-DhcpServerv4Reservation** = creates a DHCP reservation, ensuring a specific device (by MAC address) always receives the same IP address.
- **PrintManagement module (Get-Printer / Add-Printer / Get-PrintJob)** = manage print servers and print queues directly from PowerShell — a scriptable alternative to the Print Management console, useful for auditing a large printer fleet across multiple sites.

    - **Get-Printer** = lists installed printers and their driver/port/status.
    - **Add-Printer -ConnectionName \\PrintServer\PrinterName** = connects a client to a shared network printer.
    - **Get-PrintJob** = shows current jobs queued for a specific printer — the quick first check for a 'my print job is stuck' ticket.
    - **Get-PrinterDriver** = lists installed printer drivers, useful for auditing outdated/vulnerable drivers across a fleet of mixed printer models.
- **MicrosoftTeams module (Connect-MicrosoftTeams / Get-CsOnlineUser)** = manage Microsoft Teams settings and Teams-enabled users, including calling policies and phone number assignments — used alongside (but separately from) the Microsoft.Graph module, since some Teams-specific settings aren't yet exposed through Graph.
    - **Get-CsOnlineUser** = retrieves a user's current Teams-related settings (enterprise voice status, assigned policies).
    - **Get-CsTeamsCallingPolicy / Grant-CsTeamsCallingPolicy** = view or assign calling policies controlling which calling features a user has access to.
- **PnP.PowerShell (Connect-PnPOnline / Get-PnPList)** = a widely-used community/Microsoft-supported module for deep SharePoint Online and OneDrive administration — capable of far more granular site, list, and permission management than what's exposed through Microsoft Graph alone, especially for legacy SharePoint-specific features.
    - **Connect-PnPOnline -Url \<siteurl\> -Interactive** = authenticates against a specific SharePoint site collection.
    - **Get-PnPList / Get-PnPListItem** = retrieve SharePoint lists and their items, useful for bulk data audits or migrations.
- **PSWindowsUpdate module (Get-WindowsUpdate / Install-WindowsUpdate)** = a popular community module that exposes Windows Update as scriptable PowerShell cmdlets, since there's no fully-featured native module for triggering/installing OS updates directly — commonly used for one-off patch scripting or Task Scheduler-driven update automation outside of WSUS/Intune's normal update rings.
    - **Get-WindowsUpdate** = lists available (not yet installed) updates.
    - **Install-WindowsUpdate -AcceptAll -AutoReboot** = installs all available updates and reboots automatically if required.
- **Hyper-V module (Get-VM / New-VM / Start-VM / Checkpoint-VM)** = manage Hyper-V virtual machines directly from PowerShell — the Windows-native equivalent in purpose to `virsh` on Linux/KVM.
    - **Get-VM** = lists virtual machines and their state (Running/Off/Saved) on a Hyper-V host.
    - **New-VM -Name -MemoryStartupBytes -NewVHDPath -NewVHDSizeBytes** = provisions a new virtual machine.
    - **Start-VM / Stop-VM / Restart-VM** = control VM power state, directly analogous to `virsh start/shutdown/reboot`.
    - **Checkpoint-VM** = creates a point-in-time snapshot of a VM, the Hyper-V equivalent of `virsh snapshot-create-as`.
    - **Get-VMHardDiskDrive / Resize-VHD** = inspect and resize virtual hard disks attached to a VM.
- **PSReadLine** = the module responsible for PowerShell's modern interactive console experience: command history search, syntax highlighting, tab completion, and multi-line editing. Ships built into modern PowerShell by default, but its behavior can be customized (e.g., enabling predictive IntelliSense) via `Set-PSReadLineOption`.
    - **Set-PSReadLineOption -PredictionSource History** = enables inline command suggestions based on your prior command history as you type, similar in spirit to a shell history-based autosuggestion plugin in Bash/Zsh.
- **$PROFILE** = an automatic variable holding the path to the current user's PowerShell profile script — a script that automatically runs every time a new PowerShell session starts, used to set up custom functions, aliases, module imports, and prompt customizations. The direct PowerShell equivalent of Bash's `~/.bashrc`.

```
if (-not (Test-Path $PROFILE)) { New-Item -Path $PROFILE -ItemType File -Force }
notepad $PROFILE
```

    - There are actually up to four distinct profile files (current user/current host, current user/all hosts, all users/current host, all users/all hosts) — `$PROFILE` by itself refers only to the most specific one (current user, current host).

- **Pester** = the standard unit-testing framework for PowerShell, used to write automated tests that validate a script or module behaves as expected — increasingly used in sysadmin workflows for configuration validation testing (writing a Pester test that asserts a server's actual state matches an expected baseline, e.g., 'the Spooler service should be Running').

```
Describe "Web Server Health" {
    It "Spooler service should be running" {
        (Get-Service Spooler).Status | Should -Be "Running"
    }
}
```

- **Desired State Configuration (DSC)** = a declarative configuration management platform built into PowerShell, conceptually similar in purpose to Ansible or Puppet on Linux — you define the desired end-state of a machine's configuration in a `.ps1` configuration script, compile it into a `.mof` file, and apply it to enforce/correct configuration drift automatically.
    - **Configuration** = the keyword used to define a DSC configuration block, containing one or more resource blocks (File, Service, WindowsFeature, Registry, etc.) describing the desired state.
    - **Start-DscConfiguration** = applies a compiled configuration to a target node.
    - **Get-DscConfiguration / Test-DscConfiguration** = check a node's current applied configuration, or test whether it's currently in compliance with the desired configuration without making changes.
- **WebAdministration module (Get-Website / New-Website / Get-WebBinding)** = manages Internet Information Services (IIS) directly from PowerShell — sites, application pools, and bindings, the Windows-native equivalent in purpose to editing nginx/Apache config on Linux, but exposed as structured cmdlets and a dedicated `IIS:` PSDrive instead of flat text config files.

```
Get-Website
New-WebBinding -Name "Default Web Site" -Protocol https -Port 443 -HostHeader
"portal.contoso.com"
Restart-WebAppPool -Name "DefaultAppPool"
```

    - **Get-Website / New-Website / Remove-Website** = list, create, or remove an IIS site.
    - **Get-WebBinding / New-WebBinding** = list or create host header/port/protocol bindings for a site.
    - **Get-WebAppPoolState / Restart-WebAppPool / Stop-WebAppPool / Start-WebAppPool** = manage IIS application pool lifecycle — restarting an app pool is IIS's equivalent of `systemctl restart` for a specific hosted application, without needing to restart the entire IIS service.
    - **Get-ChildItem IIS:\Sites** = browsing IIS configuration directly through the `IIS:` provider drive, the same provider-based pattern used by `Cert:` and `WSMan:`.
- **Failover Clustering module (Get-Cluster / Get-ClusterNode / Move-ClusterGroup)** = manage Windows Server Failover Clustering (WSFC) — the high-availability framework underlying clustered file servers, SQL Server Always On Availability Groups, and Hyper-V clusters.
    - **Get-Cluster** = shows basic cluster identity and configuration.
    - **Get-ClusterNode** = lists nodes in the cluster and their state (Up/Down/Paused).
    - **Get-ClusterGroup** = lists clustered roles/resource groups and which node currently owns each one.
    - **Move-ClusterGroup -Name \<role\> -Node \<node\>** = manually fails over a clustered role to a specific node — used both for planned maintenance (draining a node before patching) and for manual recovery after an unexpected failover.

- **UpdateServices module (WSUS) — Get-WsusServer / Get-WsusUpdate** = manages a Windows Server Update Services (WSUS) server from PowerShell — approving/declining updates, checking client reporting status, and auditing update compliance across a fleet centrally managed through WSUS rather than cloud-based Windows Update for Business rings.
    - **Get-WsusServer** = connects to and returns the WSUS server object used by all other WSUS cmdlets.
    - **Get-WsusUpdate** = lists updates known to the WSUS server, filterable by approval and installation status.
- **File Server Resource Manager (FSRM) — New-FsrmQuota / New-FsrmFileScreen** = manages storage quotas and file screening (blocking specific file types from being saved) on Windows file servers — the Windows-native equivalent in purpose to disk quota enforcement on a Linux file server, though implemented as a distinct add-on role rather than a filesystem-level feature.
    - **New-FsrmQuota -Path \<path\> -Size \<bytes\>** = sets a storage quota limit on a specific folder.
    - **New-FsrmFileScreen -Path \<path\> -Template \<templatename\>** = blocks specific file types (e.g., executables, media files) from being saved to a monitored folder based on a predefined screening template.
- **ImportExcel (community module)** = an extremely popular third-party module (installable via `Install-Module ImportExcel`) that reads and writes native .xlsx Excel files directly, with no dependency on Excel itself being installed — used constantly for generating polished, multi-sheet, formatted reports directly from a script (far more capable than plain `Export-Csv` for anything destined for a business audience).

```
Get-ADUser -Filter * -Properties Department | Export-Excel -Path
"C:\Reports\Users.xlsx" -AutoSize -TableName Users -WorksheetName "All Users"
```

    - **-AutoSize** = automatically sizes columns to fit their content.
    - **-ConditionalFormat / -ChartType** = the module also supports adding conditional formatting rules and native Excel charts directly from PowerShell, without needing Excel automation (COM) at all.
- **dbatools (community module)** = a large, widely-adopted community module purpose-built for SQL Server database administration from PowerShell — covering backup/restore, migration, performance checks, and configuration auditing across dozens of specialized cmdlets, filling a gap left by Microsoft's own smaller native SqlServer module.
- **PowerShellGet (module management)** = the module that manages *other* modules — it's what powers `Install-Module`/`Update-Module` covered in the Scripting section.
    - **Find-Module** = searches a configured repository (like the PowerShell Gallery) for a module by name before installing it.
    - **Register-PSRepository** = adds a custom or internal module repository (e.g., a private company feed) as a source PowerShellGet can search and install from.
    - **Publish-Module** = uploads a module you've written to a repository so others can install it with `Install-Module`.
- **PSScriptAnalyzer (module)** = a static-analysis linter for PowerShell code. `Invoke-ScriptAnalyzer -Path .\script.ps1` flags style violations, common bugs, and unapproved-verb warnings before the script ever runs — the PowerShell equivalent of running `shellcheck` against a Bash script.
- **Microsoft.PowerShell.SecretManagement / SecretStore** = a standardized way to store and retrieve secrets (passwords, API keys, tokens) from a script without ever hardcoding them in plaintext. `SecretManagement` provides the common cmdlets (`Get-Secret`, `Set-Secret`); `SecretStore` is Microsoft's default local, encrypted vault that plugs into it.
- **SqlServer (module)** = Microsoft's official module for administering SQL Server directly from PowerShell. `Invoke-Sqlcmd -ServerInstance <server> -Query "SELECT ..."` runs a T-SQL query and returns the results as PowerShell objects, without needing `sqlcmd.exe` or SSMS.
- **VMware PowerCLI (community module)** = the widely-adopted community module for managing VMware vSphere/ESXi environments from PowerShell — `Connect-VIServer`, `Get-VM`, `New-VM`, and `Start-VM` mirror the same Verb-Noun patterns used by the Hyper-V module, just against a VMware backend instead.

- **SharePoint Online Management Shell (Microsoft.Online.SharePoint.PowerShell)** = Microsoft's official tenant-level administration module for SharePoint Online, distinct from the site-level PnP.PowerShell module covered above. `Connect-SPOService` connects to the SharePoint admin center, and `Get-SPOSite`/`Set-SPOSite` manage site collections at the tenant level.