ChatGPT

The Engineering Framework for Modern PowerShell Development

I. Core Philosophy: From Scripting to Engineering

Moving Beyond Monolithic Scripts: Traditional PowerShell scripts often start small but grow into unwieldy, *monolithic* blocks of code as they handle more use cases [1]. Such monoliths become brittle in enterprise settings – a change in one part can unexpectedly break another, and debugging a 1000-line script is daunting. This is the *Modification Lifecycle problem*: every update carries high risk because all logic is interdependent. By contrast, **modular design** breaks functionality into discrete units, making changes safer and easier to understand. A Reddit discussion illustrates this with a user provisioning script: by refactoring it into functions (create AD account, create mailbox, assign groups, etc.), the main script becomes a clear sequence of steps, and each sub-task can be updated in one place without affecting others [2] [3]. In short, modular code is easier to maintain – *"a huge monolithic codebase can become like a house of cards,"* whereas breaking it into pieces yields stability [4].

"LEGO Brick" Modularity and Determinism: Treating PowerShell scripts as engineered software means designing them like LEGO bricks – small, deterministic pieces that fit together. Each piece should have a single responsibility and well-defined inputs/outputs. This approach brings multiple benefits:

- **Precision Debugging:** When each function does one thing, you can test and debug it in isolation. It's immediately clear which component failed. In the earlier example, if onboarding fails at "create folder", you debug only that function, not the whole script [5] [6].
- **Confident Change and Reuse:** Modular functions can be updated or replaced without rewriting the world. If a file server path changes, only the folder-creation function needs an update [6]. Meanwhile, other scripts can reuse that function, avoiding duplicate code (the DRY principle: *Don't Repeat Yourself* [7]). Organizations often achieve this by publishing internal modules so that common functions are available to all teams.
- **Predictability through Determinism:** Each module, given the same inputs, should produce the same outputs without hidden side-effects. This determinism makes automation reliable. In testing terms, a deterministic function is far easier to validate with predictable outcomes.

The Five Module Categories – A Deterministic Framework: Modern PowerShell development can be framed by five distinct categories of modules. Each category has a specific **purpose**, expected **characteristics**, and strict **guardrails** to ensure deterministic, maintainable behavior:

1. **Data Acquisition** – *Purpose:* Reading or collecting external data and system state. These functions connect to external sources (files, APIs, AD, etc.) but **do not modify anything**. *Characteristics:* They should implement timeouts, retries, and robust input validation, since they deal with unreliable external sources. *Guardrails:* Absolutely **no state changes** (no writes to disk/DB/etc.) – treat them as pure readers. Any network or credential operations should be isolated and secure. Given the same input and external state, they return the same output (idempotent reads). *Example:* `Get-UserData`

`-Id 123` might call an API and return a user object, but it never changes the user; it only retrieves information (and perhaps caches or logs minimal metadata).

2. **Data Transformation** – *Purpose:* Pure functions that take input data and transform it into a new format or structure. *Characteristics:* No side effects or I/O; simply compute and return results. These are highly deterministic and easy to unit test exhaustively. *Guardrails:* They should not access external systems or modify global state – given identical input, they must produce identical output. Use explicit schemas or objects for inputs/outputs to enforce contracts. Aim for near 100% code coverage on these since they are the core logic. *Example:* `Convert-CsvToJson` reads CSV text (passed in as a string or object) and returns JSON text – it doesn't read files itself or write output to disk, making it a purely functional transformer.

3. **State Change** – *Purpose:* Functions that **Create, Update, or Delete** resources (on disk, in AD, in cloud, etc.). These effect changes in the environment. *Characteristics:* They should strive for *idempotence*: if you run them twice with the same input, the second run should detect nothing to change (if possible). They must support PowerShell's `-WhatIf` and `-Confirm` common parameters to preview or confirm changes [8] [9]. Incorporate robust error handling, and even rollback or compensation logic for complex changes (e.g., if step 3 fails, undo steps 1–2). *Guardrails:* **Preconditions and Postconditions** should be clearly defined – e.g., check that the target exists or is reachable before attempting a change, and verify the outcome after. Use `SupportsShouldProcess` with an appropriate `ConfirmImpact` level so that dangerous actions require user confirmation [10]. Also implement `-WhatIf` to allow dry-run mode that *only logs intended actions without performing them* [9]. Follow PowerShell's guidelines for ShouldProcess usage to provide a safety net (nothing is scarier to a user than running a script not knowing what it will delete – `-WhatIf` alleviates that anxiety [8]). *Example:* `Set-UserHomeFolder -User X -Path Y` might create a directory and set permissions. It should support `-WhatIf` to just report "Would create folder Y for X" [9], and perhaps have a `-Force` to bypass confirmation if `ConfirmImpact` is high [11].

4. **Configuration/Validation** – *Purpose:* Validating that the environment or inputs meet certain configuration standards, or enforcing those standards. In some cases, these modules don't change state but ensure that *someone* (or some system) has done so according to policy. They assert invariants and compliance. *Characteristics:* Often declarative – you provide a desired state description or rules, and the function/test checks reality against it. They may leverage rule engines or policy frameworks (e.g. **PSRule** for infrastructure as code validation). *Guardrails:* They should **fail fast** on violations (clearly signaling an issue) and produce machine-readable reports (e.g. structured results or error objects) so that CI/CD pipelines can act on them. These functions contain **no orchestration or remediation logic** – they *report* compliance, they don't fix things (fixing would be a State Change concern). Also, keep them separate from actual orchestration so that you can run validation standalone. *Example:* `Test-Config -Settings .\desiredConfig.json` might read a JSON of expected settings and verify the live system matches, outputting any mismatches. A real-world illustration is **PSRule** modules that package sets of rules – for example, PSRule can validate Azure resource templates for best practices and produce pass/fail outcomes [12] [13]. PSRule is distributed as modules and uses tags like `PSRule-rules` in module metadata so it can auto-discover rule packs [14] [15]. This underscores how validation modules can be built and shared just like any other PowerShell module.

5. **Orchestration** – *Purpose:* Higher-level workflows that **sequence and coordinate** the other modules. Think of an orchestrator as a PowerShell *script* that wires the LEGO bricks together. For instance, to onboard a new user, an orchestrator might call an Acquisition function to get a free username, call a Transformation to format the account data, call multiple State Change functions to create accounts and mailboxes, then a Validation to confirm everything is correct. *Characteristics:* Orchestrators handle flow control: looping, parallel execution (when safe), conditional logic, and retry policies. They often implement **transient fault handling** (retry/backoff) at a higher level if lower-level functions signal a retriable error. They also gather logging and metrics from the steps to provide end-to-end observability. *Guardrails:* **No business logic in the orchestrator.** All substantive work should be delegated to the appropriate category function. Orchestrators should be "thin" in terms of logic – if you find an orchestrator calculating something complex, that portion likely belongs in a Transformation or Acquisition function. Also, orchestrators should not directly mutate state – they call State Change functions to do that – ensuring that all changes still go through the controlled, testable paths. Another guardrail is structured logging: orchestrators should use consistent logging (maybe via a logging module like PSFramework's `Write-PSFMessage`) so that every step is traceable. They should emit timing and success/failure info for each step (which is crucial when diagnosing long workflows). *Example:* `New-UserOnboarding -Username Bob -Dept Sales` might orchestrate by: calling `Find-AvailableUsername` (Acquisition), `New-ADUser` (StateChange), `New-Mailbox` (StateChange), `Set-UserHomeFolder` (StateChange), then `Test-UserConfig` (Validation). It would handle errors at each step, perhaps with rollback if something in the middle fails, and finally output a summary.

These five categories impose an **explicit separation of concerns**. By categorizing each function/module upfront, teams create clear guardrails: no sneaky side-effects in an Acquisition module, no business logic in an Orchestrator, etc. This determinism and modularity greatly improves reliability. It allows one to modify, for example, how data is retrieved (Acquisition) without risking the core logic (Transformation) or how final steps are orchestrated.

II. The Modular Development Lifecycle & Workflow

Designing PowerShell modules in an engineered, enterprise-grade way involves a structured lifecycle. We can think in phases: **Blueprint -> Build -> Test -> Package -> Deploy/Reuse**, with feedback loops. Key practices at each phase ensure we realize the benefits of modularity.

Blueprint Phase: Plan with the 5 Categories

Before writing a single line of code, decompose the **business requirement into the five categories**. This is analogous to system design in classical software engineering. In practice, read the user story or task and identify: - What data do we need and where does it come from? (This becomes candidates for *Data Acquisition* functions.) - What core logic or calculation is needed? (Likely *Data Transformation* functions.) - What changes will we make to systems? (Those will be *State Change* functions.) - What checks or validations are required to ensure compliance or correct setup? (*Configuration/Validation* functions.) - Finally, how will these pieces flow together? (The *Orchestration* script.)

Example (Blueprint): Suppose we must automate onboarding a new employee. The blueprint might outline: 1. **Acquisition:** Get the next available employee ID and email alias (from HR system or by scanning AD). 2. **Transformation:** Generate user account attributes (username, initial password, group

memberships) from input data. 3. **State Changes:** Create AD account, Exchange mailbox, home directory, etc., each as separate idempotent functions (e.g., `New-ADUserAccount`, `Enable-Mailbox`, `Setup-HomeDirectory`). 4. **Validation:** Verify the new accounts meet security policies (password complexity, group memberships, etc.) – possibly using a PSRule rule set for AD user standards. 5. **Orchestration:** A `New-EmployeeOnboard` script that ties all the above steps together in order, with proper error handling and rollback (e.g., if mailbox creation fails, perhaps disable AD account as cleanup).

By **mapping requirements to module categories upfront**, we avoid the "single script does everything" trap. We also set ourselves up to write testable units. Notably, this blueprint phase also aligns stakeholders: it's easier to discuss a design when you can say "we'll have one function to fetch data from system X, then a pure function to compute Y, ..." instead of diving into script internals.

Structure Creation: Directory Layout and Encapsulation

Once functions are identified, a consistent **project structure** keeps things organized. A common layout for a module project is:

[16] PowerShell modules are not just single files; they support a folder structure to include multiple files, tests, and docs. The module's root will typically contain: - A module manifest (`.psd1` file) containing metadata, exported functions list, required modules, etc. - A module implementation file (`.psm1` or a set of `.ps1` files) that actually dot-sources or imports all the functions. - Clearly separated folders for *Public* and *Private* functions (especially in larger modules). **Public functions** are the module's interface – these get exported via the manifest's `FunctionsToExport` or via `Export-ModuleMember` in the psm1. **Private functions** are internal helpers not exposed to users; they reside in a `Private` folder and are dot-sourced or imported inside the module so public commands can use them.

For example, a **minimal** module scaffold might look like:

```
MyModule/
├─ MyModule.psd1
├─ MyModule.psm1
├─ src/
│  ├─ Public/        # public functions (.ps1 files for each command)
│  └─ Private/       # private helper functions
└─ tests/            # Pester tests
```

(See Appendix A for scaffold examples.)

In practice, the `.psm1` can either contain all code or (better) act as a loader that dot-sources every `.ps1` in the `Public` (and maybe `Private`) folders. This way each function is defined in its own file, which simplifies source control and collaboration (multiple people can work on different functions in parallel). The manifest (`.psd1`) explicitly lists the public functions to export, ensuring that private helper functions aren't exposed accidentally.

Encapsulation is key: external users should only rely on the Public interface. By hiding implementation details (like `Helper-ComputeHash` existing in Private), you can refactor internals without breaking users. The **dbatools** module is a good real-world example of this principle. In dbatools, all user-facing commands live as separate `.ps1` files under a *functions* directory, and there's a `SharedFunctions.ps1` (full of internal helper functions) that is dot-sourced into the module but not exported [17] [18]. The module manifest `dbatools.psd1` exports only the functions meant for users, and it's maintained as new commands are added [18]. This structure has allowed dbatools (with nearly 700 functions) to scale – contributors can add new command files easily, and the team can modify internal shared logic in one place [19].

Beyond Public/Private split, enterprise modules may include additional folders: - `examples/` for sample usage scripts. - `docs/` or at least a README.md for documentation. - A `build/` folder if you have build scripts (e.g., psake tasks or other automation to package the module). - Continuous Integration config, e.g., a `.github/workflows/` folder for GitHub Actions, or Azure Pipelines YAML in the repo root.

Establishing this scaffold at project start ensures all team members put things in the right place and follow a convention. It also ties into tooling – for instance, if you use Plaster (a templating tool) or PSModuleDevelopment to generate a module project, they will create a structure like above automatically.

Development Order: Inside-Out Coding (Logic First)

When implementing the module, a recommended order of attack is: **start with the core logic (Transformation and Validation), then Acquisition and StateChange, and finally the Orchestrator**. This is akin to writing library code first, scripts second.

1. **Write pure transformations early:** Since these have no external dependencies, you can develop and unit test them immediately. They often represent the business logic. For example, implement the `New-UserAttributes` (Transformation) that takes input from HR and outputs an object for AD creation. Writing it first clarifies what input it needs and what output it produces, which in turn informs what data acquisition functions must retrieve and what the state change functions will consume.

2. **Define configuration/validation logic next:** If there are any defined policies or desired state checks, writing these early serves as a form of "executable specification." For instance, if password policy is critical, you might write a `Test-PasswordCompliance` that fails if rules not met. This can guide development of the state-change function that sets passwords (it must call or adhere to `Test-PasswordCompliance`).

3. **Implement Data Acquisition and State Change:** With clear input/output contracts from the above, now build the functions that talk to external systems. Because you know exactly what data format they should return or accept, you reduce thrash. Implement `Get-NextUsername` (Acquisition) to fetch an ID from HR DB, and `New-ADUserAccount` (StateChange) to create an AD account given the attributes. Aim to make these idempotent or at least safe to re-run. For example, `New-ADUserAccount` could check if the user exists first (with the same properties) to avoid duplicates – this could be implemented with a `-WhatIf` support to simulate the creation without committing, aiding idempotence testing [8] [9].

4. **Compose with the Orchestrator last:** Finally, write the `New-EmployeeOnboard` orchestrator that calls all the above in sequence. By doing this last, you ensure that all building blocks are available and tested in isolation. The orchestrator should be mostly gluing function calls together and handling flow. It might also handle cross-cutting concerns like logging the start and end of each step, measuring durations, etc., but it does not implement the business rules itself. If while writing this script you find you need a new piece of logic, resist coding it inline – instead create a new function (in the appropriate category) for it. This discipline keeps orchestration scripts very clean and readable: you can glance at the orchestrator and see the high-level workflow (almost like pseudo-code calling each major step) [20].

This development order also aligns with testing (discussed next): by the time you write the orchestrator, the lower-level functions ideally already have passing unit tests, so if something fails in the orchestrator integration test, you know which component might be at fault.

Testing Strategies by Category (Pester to the Rescue)

Testing is non-negotiable in an engineered approach. **Pester**, PowerShell's ubiquitous test framework, is used to write and run tests for each module component. A key advantage of the categorical design is that it naturally guides how to test different pieces:

- **Transformation Functions:** These are the easiest to unit test. Since they have no side effects, you can call them with sample inputs and assert that the outputs match expected values. Use **Pester** `Describe/It` blocks to create unit tests covering normal cases, edge cases, and error cases. Aim for exhaustive coverage here (if logic branches exist, each should be tested). For example, given an input object, does `Convert-ToCsvRecord` produce the correct CSV string? You can craft a dummy input and expected output and use `| Should -Be` assertions [21] [22]. See Appendix B for an example Pester test of a pure function.

- **Data Acquisition Functions:** These call external systems, which we do *not* want to actually hit during unit tests (tests should run in isolation and not depend on live systems). Pester provides a powerful **Mocking** capability to handle this [23] [24]. The strategy: *mock the external calls*. For instance, if `Get-UserData` internally calls `Invoke-RestMethod` to an API, your test can use `Mock Invoke-RestMethod -ReturnValue $fakeResponse` to simulate a response [25]. Then assert that `Get-UserData` returns the expected object given that fake API response. Also test failure paths – e.g., mock `Invoke-RestMethod` to throw an exception and ensure `Get-UserData` properly handles or surfaces the error. The goal is to validate *our code's reaction* to external data, without needing the actual external system. Pester's mocking and `Assert-MockCalled` can verify that your function attempted the expected calls (e.g., ensure `Invoke-RestMethod` was called with the correct URL) in addition to output validation.

- **State Change Functions:** These are tricky to test because they perform real actions. The approach is twofold: use **WhatIf/Confirm** and **Mocking**. First, ensure the function supports `-WhatIf` and `-Confirm` (we consider that part of the implementation). You can then test that in `-WhatIf` mode it **does not** actually change anything but reports the correct message. For example, `Remove-FileSecure -Path X -WhatIf` should output a `What if:` message without deleting the file [26]. Pester can capture that output and confirm it contains the expected text. Second, for unit tests, treat calls to external cmdlets (like `New-ADUser` or `Remove-Item`) as external and mock them. In

a test, you might do `Mock Remove-Item { $null } -Verifiable` and then call your function with `-Confirm:$false` (to skip any prompts), and finally use `Assert-MockCalled Remove-Item -Times 1` to ensure it would have invoked the remove operation [27]. Essentially, we test that our function *attempts* the right changes under the right conditions. Full integration tests (if allowed in a non-prod environment) could be a separate layer – but for CI, we rely on these simulated tests. Additionally, you can use Pester to test idempotence: run the state function twice in a test, with the second run perhaps mocked to simulate "already exists" and ensure it handles that gracefully (e.g., second run does nothing or returns a specific result indicating no change).

- **Configuration/Validation Functions:** These should be tested with both compliant and non-compliant scenarios. For a validation that outputs a report or `$true/$false`, feed it known good input and verify it passes, and feed a bad input to verify it fails (and produces the correct error or report). If using a rule engine like PSRule inside, you might not unit-test the *engine* but you should test that your rules (.Rule.ps1 or similar) fire correctly. For example, if you wrote a PSRule that "All VMs must have tagging", you can use a small dummy object in a Pester test and run `Invoke-PSRule` on it, then assert that the outcome has a fail for that rule. Another approach is to treat each rule as a small function (if possible) and test it similarly. The key is verifying the logic catches what it should catch. Also test the output format – e.g., if your `Test-Config` function is supposed to output an object or throw an exception on failure, confirm it does so.

- **Orchestrators:** Testing orchestration is more akin to integration testing. The orchestrator calls many pieces; in unit tests, we **mock each of those pieces**. Essentially, in a Pester test for the orchestrator, you would mock every underlying function (the ones you wrote in other categories) to avoid actually doing heavy work. Then you call the orchestrator and use `Assert-MockCalled` to verify it invoked sub-functions in the correct order or with expected parameters. For example, if `New-EmployeeOnboard` should call `New-ADUserAccount` then `Enable-Mailbox`, you can do something like: `Mock New-ADUserAccount { "AD ok" } -Verifiable; Mock Enable-Mailbox { "Mailbox ok" } -Verifiable; ... Invoke the orchestrator ...; Assert-MockCalled New-ADUserAccount -Times 1; Assert-MockCalled Enable-Mailbox -Times 1;`. This doesn't test the internals of those functions (which have their own tests) – it tests the *wiring and flow*. You may also simulate failures: e.g., mock `Enable-Mailbox` to throw, and verify the orchestrator handles it (maybe it calls a rollback function or returns an error status). Pester's ability to mock functions in memory makes this straightforward. Essentially, you're testing that "if all components work, the orchestrator calls them properly," and "if a component fails, the orchestrator responds properly."

Pester is extremely well-suited for these patterns and is deeply integrated into modern PowerShell dev. In fact, **Visual Studio Code's PowerShell extension** recognizes Pester tests and provides CodeLens links like "Run Tests" above each `Describe` block for quick execution [28]. This tight VSCode integration means you can run and debug tests with one click, accelerating TDD (test-driven development). Moreover, Pester can output results in NUnit XML or other formats, which CI systems like Azure DevOps, Jenkins, or GitHub Actions can ingest to display test results [29] [30]. Pester's own documentation notes how it "integrates nicely with TFS, Azure DevOps, GitHub, Jenkins and other CI servers" to automate testing in pipelines [31].

In summary, **each category has a matching test strategy** (unit tests for pure functions, mocked tests for external interactions, etc.). By following this, you achieve near-complete coverage. (Appendix B provides simple examples of Pester tests for a few categories as a reference.)

Notably, we should also run **static analysis** on our code as part of testing. The primary tool is **PSScriptAnalyzer**, which catches common issues and enforces style and best practices. In fact, the PowerShell Gallery automatically runs PSScriptAnalyzer on submitted modules and rejects those with serious issues [32]. It's wise to run it locally (or in CI) using `Invoke-ScriptAnalyzer -Recurse . -Severity Warning` to catch problems early [33]. This includes everything from syntax errors to potential security issues and style consistency. A clean static analysis (or documented exceptions) is often part of the "definition of done" for a PowerShell module in enterprise. We integrate this in CI, as described below.

Storage & Deployment: Versioning and Private Feeds

After development and testing, we need to **package and publish** the module for use. In an enterprise, you often **version your modules** and distribute them via **private repositories** (so that only your org can access them, and you have control over updates). Key considerations:

- **Module Versioning:** Use semantic versioning (semver) for your module (e.g., `1.4.0`, `1.5.0` for new features, `2.0.0` for breaking changes). PowerShell modules support semantic versioning inherently [34]. The version is specified in the module manifest (`ModuleVersion` field), and it's how PowerShell distinguishes updates. Semver discipline is important: increase the major version if you make breaking changes to function interfaces or behavior; minor for new backward-compatible features; patch for bug fixes. This allows users (and CI systems) to handle updates appropriately. Internal galleries can enforce or at least encourage using the latest stable version.

- **Private Repositories (Feeds):** Instead of publishing to the public PowerShell Gallery, enterprises use private feeds. Options include **Azure Artifacts** (part of Azure DevOps), **GitHub Packages**, **Azure Container Registry (as a module feed)**, ProGet, Nexus, etc. With the new **PSResourceGet** module (the replacement for PowerShellGet), publishing to private feeds is greatly streamlined. For example, Azure Artifacts can be used as a NuGet-based PowerShell repository [35]. You would:

- Register the repository on developer machines/CI using `Register-PSResourceRepository` with the feed URL and credentials (often using a PAT token for Azure DevOps or a GitHub PAT) [36] [37].
- Use `Publish-PSResource -Path <pathToModule> -Repository <FeedName> -Credential <cred>` to publish the module package to the feed [38]. (Under the hood, this creates a NuGet package (.nupkg) of your module and pushes it – PSResourceGet automates a lot, including versioning and dependency handling.)

For instance, Microsoft's docs show publishing a module to Azure Artifacts with one command once the repo is registered [39] [40]. Similarly, PSResourceGet 1.1+ supports publishing to Azure Container Registry (ACR) as a repository – effectively letting you use ACR to host PowerShell modules as artifacts [41] [42]. In either case, the process is: set up credentials (often with the SecretManagement module for secure storage

[43] [44]), register the repository, then publish. After publishing, others can install the module via `Install-PSResource -Name MyModule -Repository MyFeed`.

- **Handling Dependencies:** If your module depends on others (e.g., perhaps you leverage the **PSFramework** module for logging capabilities), specify that in your manifest (`RequiredModules`). When consumers install your module from the feed, PSResourceGet will auto-resolve and pull required modules (from either the same feed or PSGallery if allowed) [45]. This ensures no missing dependency at runtime. In private environments, sometimes you even mirror public dependencies internally – e.g., host a copy of PSFramework or Pester in your private feed so that your systems don't always reach out to the public gallery.

- **Versioning Strategy in CI:** Typically, you'll update the module version for each release. Some teams auto-increment the version in CI (for example, on each merge to main, bump patch version). Others manually update for significant releases. Either way, use Git tags or release notes to mark versions. It's useful to maintain a **CHANGELOG** so that internal users know what changed in each version (especially if something breaks).

- **Private vs Public Balance:** Many organizations keep most modules internal but might open-source some general ones. If you ever intend to publish publicly, ensure no sensitive info in code or metadata. Even for private modules, treat them with the same rigor (licenses, documentation) – because internal users are your customers. A common approach is an **internal PowerShell gallery** (like an Azure Artifacts feed or a simple file share registered via `Register-PSResourceRepository`). Some companies even set up an internal mirror of the PowerShell Gallery, promoting vetted modules for use. Appendix D gives a quick checklist for internal gallery usage and version management.

In summary, **Storage/Deployment** is about treating modules as first-class artifacts: versioned, published to a repository, and retrieved by users/CI pipelines as needed. Gone are the days of copying scripts via SMB share – instead, an engineer can run `Install-PSResource -Name MyModule -Repository InternalRepo` and get the latest approved code, just like grabbing a package from NuGet. This not only simplifies distribution but also encourages reuse (why write it again when it's in the repo?) and governance (you can enforce code quality by controlling what gets into the feed).

III. Tooling & Ecosystem for Modern PS Development

Modern PowerShell engineering benefits from a rich ecosystem of tools. Adopting the right editors, scaffolding utilities, linters, and CI/CD integrations turbocharges productivity and enforces quality.

Editor and Extensions: VS Code + PowerShell Extension

Visual Studio Code is the de-facto standard editor for PowerShell in 2025. With the **PowerShell Extension** (maintained by Microsoft), VS Code becomes a full-fledged PowerShell IDE. Key features relevant to our framework: - **IntelliSense and Syntax Checking:** The extension provides smart completions for cmdlets, parameters, etc., and flags syntax errors on the fly. - **Integrated Console:** You get a built-in PowerShell terminal that automatically imports your module in development for testing. You can run scripts or functions with a keystroke and see results immediately. - **Debugger:** Breakpoints, step-through debugging,

variable watchers – all available, even for PowerShell scripts. This is invaluable when diagnosing an orchestrator script or complex function logic. - **Pester Test Integration:** As mentioned, the extension detects Pester tests. It adds CodeLens above tests ("Run Test" / "Debug Test" links) [28], so you can execute a single test or a whole file directly in the editor and get results in the output window. This lowers the friction to continually run tests during development. - **EditorConfig/Formatting:** The extension supports PSScriptAnalyzer rules for formatting. Many teams include a settings file so that when you save, the code auto-formats (indentation, spacing) to a standard style. This keeps code style consistent across contributors. - **Snippets:** The extension offers snippets (shortcuts) for common PowerShell constructs. For example, typing `switch<tab>` might stub out a switch statement, or `function<tab>` yields an advanced function template with `[CmdletBinding()]` ready. Some are built-in, others you can add. This speeds up writing boilerplate like parameter definitions or comment-based help sections. - **Git Integration:** While not PS-specific, VS Code's Git UI helps with source control – seeing diffs of your PowerShell files and making commits is seamless in one interface.

In essence, VS Code with the PowerShell extension provides a **"great experience for writing PowerShell code," including features that improve how you write, run, and debug Pester tests** [46]. It's lightyears beyond the old PowerShell ISE.

Additionally, for testing and DevOps, some folks use IDEs like **Visual Studio** (with PoshTools) or **JetBrains Rider** (supports PowerShell via a plugin), but VS Code remains the most widely used, cross-platform option, supported officially.

Module Scaffolding Tools: Plaster, PSModuleDevelopment, and Sampler

Bootstrapping a new module project by hand can be tedious – creating folders, manifest, tests, etc. Scaffolding tools help enforce standards and save time by generating that structure automatically:

- **Plaster:** A template-based file and project generator for PowerShell. Plaster can ask you questions (module name, author, etc.) and produce a ready-to-go module folder. For example, using a Plaster template, you can scaffold a module that already has the `Public`/`Private` folders, an empty Pester tests folder, a README, and build scripts. According to its documentation, *"Its purpose is to streamline the creation of PowerShell module projects, Pester tests, DSC configurations, and more"* [47]. Essentially, it's like Yeoman or Cookiecutter but for PowerShell [48]. Plaster was originally from Microsoft and now community-maintained. Many official samples and community modules include Plaster templates. For instance, you could run something like `Invoke-Plaster -TemplatePath NewModule.plaster.xml -Destination C:\Dev\MyModule` and after answering prompts, get a complete scaffold.

- **PSModuleDevelopment (PSMD):** A newer tool by Friedrich Weinmann (author of PSFramework) which takes scaffolding further. It shares DNA with the Sampler project and emphasizes customization. *"PSModuleDevelopment is Fred's take on module scaffolding, and it shares a lot of DNA with Sampler. It can scaffold PowerShell modules and more..."* [49]. The key feature: **custom templates**. PSMD allows you to create your own company-specific module template with your standards (folder names, pipeline setup, etc.). Unlike a static Plaster template, PSModuleDevelopment is built from ground up to let you define and reuse templates easily [50] [51]. For example, you can create a template that includes an Azure DevOps pipeline YAML, a specific README format, and even some example functions. Then your team can use `Invoke-PSMDTemplate -TemplateName`

`"OrgModuleStd" -Name "NewModule"` to scaffold a new module that adheres to your internal standards in seconds [52]. This addresses the scenario where out-of-the-box templates (Plaster or Sampler's defaults) weren't matching a company's needs [53] [54]. With PSModuleDevelopment, teams can maintain a **template repository** (which PSMD calls a template store) – effectively version-controlling the scaffold itself and sharing it [55] [56]. This ensures consistency: every new module begins life with the correct structure, build files, and settings (no forgetting to include the license file or code of conduct, for example).

- **Sampler:** Mentioned in passing – Sampler is a project that was popular for scaffolding especially CI pipelines for modules (by Gael Colas and others). PSModuleDevelopment was influenced by Sampler but adds more flexibility in template customization [49]. Sampler provides a well-thought-out module template with build scripts (Invoke-Build based), Pester integration, code coverage, etc., all wired up. If you use Sampler, you essentially clone its template and then modify. It's great for quickly getting CI-ready modules.

In practice, you choose one of these tools to standardize how modules are created. If your org is just starting, Plaster might be simplest – use an existing template or craft one that fits your needs (maybe tweak open-source ones like the PSDscResources example template). For power users with very specific workflows, PSModuleDevelopment plus custom templates can ensure every project has identical structure and no one "forgets" to include the new standards. The benefit is consistency and time saved – no one bikesheds the project structure each time; it's decided and automated.

Build Automation: psake and Invoke-Build

For larger projects and CI/CD, it's common to have a **build script** that can perform tasks like running tests, updating manifest versions, packaging the module, etc. Instead of writing raw PowerShell scripts for each task, the community uses build automation tools: - **psake:** A PowerShell build automation tool akin to Make or Rake. It lets you define "Tasks" with dependencies, so you can say, for example, that the "Test" task depends on the "Build" task, which depends on "Clean" task, etc. psake uses PowerShell syntax – no XML build files – thus avoiding the "angle-bracket tax" of tools like MSBuild [57]. In a psake script, you write simple PowerShell functions (tasks) and then run `Invoke-psake` to execute them with dependency order. For instance, you might have a `Task Clean { remove-item .\Output -Recurse -Force }`, `Task Build -depends Clean { New-ModuleManifest ... }`, `Task Test -depends Build { Invoke-Pester }`, etc. psake makes it easy to orchestrate these steps and reuse them. It's pronounced "sake" (like the drink) and has been around a long time, used by many projects (Chocolatey's build uses psake, for example) [58] [59]. It's available on PowerShellGallery (`Install-Module psake`) and is often integrated into pipelines (just a matter of running the psake script in a CI job). One nice aspect: because psake tasks are just PowerShell, you can use any .NET or system call as needed in your automation with full PowerShell power [60].

- **Invoke-Build:** Another popular build tool by Roman Kuzmin. It is conceptually similar to psake (tasks with dependencies), and some find its syntax or performance better. The GitHub says *"Invoke-Build is a build and test automation tool which invokes tasks defined in PowerShell scripts. It is similar to psake but arguably easier to use."* [61]. Both serve the same purpose; it's often personal/team preference which to use. For instance, some templates (like Sampler) use Invoke-Build with predefined tasks for test, lint, format, publish, etc. If you see a `build.ps1` or `tasks.ps1` in a module repo, it's likely using one of these frameworks.

Why use these? They codify the workflow steps so that running them is consistent across environments. In CI, you might simply call `Invoke-Build Test` to run tests, which under the hood does the setup then calls Pester. Locally, a dev can do the same. This eliminates manual steps ("oh I forgot to run the analyzer before pushing"). Instead, one command does it all. It's an embodiment of *scripts as code* for the build itself.

A minimalist approach might skip psake and just use a simple script, but as complexity grows (multiple tasks, conditionally skipping some in CI vs local), psake/Invoke-Build shine.

For example, you could have a psake script where `Task Package -depends Test { Publish-PSResource ... }` to automatically publish the module if tests passed and you invoked the Package task. This can be triggered in CI on a release branch. The consistency helps avoid human error in release processes.

Continuous Integration/Delivery (CI/CD): Automated Testing and Publishing

Integrating with CI/CD ensures every change is validated and modules are continuously deliverable. Given our earlier discussions, a typical CI pipeline for a PowerShell module might do: 1. **Install dependencies** (if any, e.g., perhaps your module relies on another internal module – you'd install it from the feed in CI, or if using PSFramework or Pester 5, ensure those are present). 2. **Static analysis**: Run PSScriptAnalyzer to catch any linter issues or styling (failing the build on any "Error" severity issues) [62] [33]. 3. **Run Pester tests**: Execute all tests and collect results. In Azure DevOps or Jenkins, you might publish results (NUnit XML) to see test reports. In GitHub Actions, you can use the output or a reporter to annotate failures. 4. **Package or publish**: If tests pass, package the module (maybe create a nupkg via `Publish-PSResource -Repository … -NoPublish` or simply zip the folder, or copy to an artifacts staging area). Optionally, publish to the internal feed if this pipeline is for continuous deployment. 5. **Artifact/Gallery deployment**: In some setups, CI produces a artifact (like a .nupkg or .zip) which then a release pipeline or manual step pushes to the gallery. Other setups go straight to publishing it when on a main branch.

Jenkins: For Jenkins, since it's script-based, one would typically use a **PowerShell plugin** or just call `pwsh.exe`. You might see a Jenkinsfile with stages like:

```
stage('Test') {
    steps {
        powershell 'Install-Module Pester -Scope CurrentUser -Force; Invoke-
Pester -OutputFile TestResult.xml -OutputFormat NUnitXml'
    }
    post { always { publishJUnit testResults: 'TestResult.xml' } }
}
```

This runs Pester and uses the Jenkins JUnit plugin to record results. Jenkins can also use the PSScriptAnalyzer module similarly (run it, perhaps archive the results or fail on error). Essentially, Jenkins integration is straightforward via its PowerShell capabilities, though you have to configure the environment (ensuring PowerShell 7 and modules are available on the agent). Pester's documentation mentions such integration and indeed many have used Jenkins + Pester successfully [31].

Azure DevOps (ADO) Pipelines: With Azure DevOps, you can use the built-in PowerShell task in YAML. Example snippet:

```
- pwsh: |
    Install-Module PSScriptAnalyzer -Scope CurrentUser -Force
    Invoke-ScriptAnalyzer -Path . -Recurse -Severity Warning,Error -OutFile
sascan.xml
  displayName: 'Static Code Analysis'

- pwsh: |
    Install-Module Pester -Scope CurrentUser -Force
    Invoke-Pester -OutputFormat NUnitXml -OutputFile TestResults.xml
  displayName: 'Run Pester Tests'

- task: PublishTestResults@2
  inputs:
    testResultsFiles: '**/TestResults.xml'
    testRunner: NUnit
  condition: succeededOrFailed()
```

This would run analysis and tests, then publish the results to the ADO test report. For publishing to a private feed (Azure Artifacts), you could add:

```
- pwsh: |
    Write-Host "Publishing module to internal feed"
    Publish-PSResource -Path .\MyModule -Repository "MyAzArtifactsFeed" -
NuGetApiKey az
  displayName: 'Publish Module'
  condition: succeeded()
```

(Here we assume `NuGetApiKey 'az'` is a dummy string required by API, and credentials are handled by the feed being pre-authenticated via environment.) Azure DevOps also offers an easier route: using a service connection for Azure Artifacts so you might not even need API keys if on Microsoft-hosted agents. The official docs walk through using Azure Artifacts with PSResourceGet, including creating a PAT and setting up a secret, etc. [36] [63] .

GitHub Actions: GitHub Actions has become very popular for PowerShell projects, especially open-source. An example GH Actions workflow (YAML) for a PowerShell module:

```
name: CI
on: [push]
jobs:
  build-test:
    runs-on: windows-latest  # (or ubuntu-latest, since PowerShell Core works
cross-platform)
```

```
    steps:
      - uses: actions/checkout@v3
      - name: Setup PowerShell
        uses: PowerShell/PowerShell-Build@v0  # optional: to ensure latest PS
      - name: Install PSScriptAnalyzer
        run: pwsh -Command "Install-Module PSScriptAnalyzer -Scope CurrentUser -
Force"
      - name: Run PSScriptAnalyzer
        run: pwsh -Command "Invoke-ScriptAnalyzer -Path ./MyModule -Recurse;
if(!$?) { exit 1 }"
      - name: Run Pester Tests
        run: pwsh -Command "Invoke-Pester -OutputFormat NUnitXml -OutputFile
TestResults.xml"
      - name: Publish test results
        uses: actions/upload-artifact@v3
        if: always()
        with:
          name: PesterResults
          path: TestResults.xml
      - name: Publish to PSGallery
        if: github.ref == 'refs/heads/main'
        env:
          PSGALLERY_API_KEY: ${{ secrets.PSGalleryApiKey }}
        run: pwsh -Command "Publish-Module -Path ./MyModule -Repository
PSGallery -NuGetApiKey $Env:PSGALLERY_API_KEY"
```

This is illustrative: it lints, tests, and on main branch pushes to PowerShell Gallery (the public gallery) – you'd adjust to your internal repository. GitHub Actions allows secure storage of secrets (like API keys). There's even a GitHub Action specifically for Pester tests that nicely formats output, but using plain PowerShell is fine. The official GitHub docs on PowerShell CI provide a simple example where they run a one-liner test in a workflow [64] [21], but real projects will run full Pester as shown.

One can also integrate **code coverage** (Pester can output coverage data), or use **CodeQL** for security scanning on PowerShell, etc., as part of CI.

The bottom line: CI ensures that every commit of your module is automatically validated by PSScriptAnalyzer and Pester (at minimum), giving the team immediate feedback if something breaks. CD (continuous delivery) can be achieved by auto-publishing modules to an internal feed after, say, a merge to a `main` or a version bump. This leads to rapid iteration cycles – a developer merges a feature, the pipeline runs tests, publishes version 1.5.0 to the internal feed, and minutes later that module is available for use in production automation. All with confidence because tests and analysis gates were passed.

(Appendix C provides a sample GitHub Actions YAML and an Azure DevOps YAML for reference.)

IV. Real-World Examples & Authoritative Resources

It's instructive to examine some **well-structured PowerShell modules** in the wild. These projects illustrate best practices in modular design, testing, and project layout:

- **dbatools:** A wildly popular open-source module for SQL Server automation (by Chrissy LeMaire and team). **dbatools** contains hundreds of functions (for migrating databases, managing backups, etc.) and is a shining example of community-driven module development. *Structure:* All functions are in the `functions/` directory as individual .ps1 files [65]. The module has a manifest `dbatools.psd1` that explicitly exports functions. There are special files like `SharedFunctions.ps1` for internal use across multiple commands [17], and `DynamicParams.ps1` to centralize all dynamic parameters logic (tab-completion etc.) [66]. This separation means any contributor adding a new command just drops a .ps1 in `functions`, and includes their function name in the manifest's FunctionsToExport (the maintainers handle that to avoid merge conflicts) [18]. The tests are in a `tests/` folder (they even test that the module's help documentation is present via `InModule.Help.Tests.ps1` [67]). dbatools emphasizes backward compatibility and multi-platform support, requiring a robust CI setup – they use Pester for everything, and had Jenkins in the past for integration tests across SQL versions. *Why it works:* The clear file structure and contribution guidelines prevent chaos despite many contributors. The module is treated like a software product with continuous testing (ensuring that adding a new function doesn't break existing ones, etc.). For example, in guidelines they encourage copying patterns from existing commands and note the project's pride in working on even older PowerShell versions and SQL Server versions [68] – which means they have to systematically test across those, something they can do thanks to their modular approach (specific functions handle old SQL quirks, etc., without complicating others).

- **PSFramework:** This module (by Friedrich Weinmann) is essentially a toolkit or framework for other module authors. It provides logging facilities, configuration management, and utility functions. *Structure:* The project is complex under the hood (even includes C# binary parts), but logically it has a `PSFramework/` folder with the PowerShell scripts and a `library/` with binary components [69] [70]. It organizes functions into sub-folders by feature (e.g., repository management commands might be under `PSFramework/Repository/Set-PSFRepository.ps1` etc.). A notable aspect is the **focus on infrastructure**: PSFramework introduces things like global configuration that any module can use (via `Get-PSFConfig` etc.), centralized logging via `Write-PSFMessage`, and pipeline optimization tools. For example, replacing Write-Verbose with Write-PSFMessage instantly adds logging to file and other destinations [71] [72]. *Why emulate:* PSFramework shows how you can build a module that is very **configurable and extensible**. It uses its own configuration system so that any setting (like log levels, or whether to enable a certain feature) can be changed by users at runtime or via config files. It also demonstrates good use of PowerShell's help system – it has many *about_* help files included (in `help/en-US/about_*.help.txt` files) covering conceptual topics (just as an enterprise module should ship with thorough help). It's well-tested and continuously integrated. If your automation project needs common utilities (like enhanced logging), you might *reuse* PSFramework rather than reinventing – it's designed to be imported as a dependency. Studying its repo also provides patterns for multi-module solutions (it actually has sub-modules like PSFramework.Logging perhaps, though distributed as one).

- **Az Modules (Azure PowerShell):** The **Az** module is Microsoft's official Azure cmdlet library. It is massive (spanning dozens of service-specific sub-modules like Az.Compute, Az.Storage). *Structure:* Az is a **roll-up module** that imports all the others. Each Azure service (Compute, Network, KeyVault, etc.) has its own module with cmdlets for that service, all under the Az umbrella. This modularization was intentional – it allows updates to individual services without impacting others, and users can choose to install only needed service modules for performance [73] [74]. The Azure team has a common pattern for each module (likely generated from Swagger specs, partially). Each module has a folder for its cmdlets, many of which are implemented in C# for performance. But what's relevant is how Microsoft structured a huge project into manageable pieces. They use a **common Az.Accounts** module on which all others depend (for auth and common functionality) [75]. By separating each service, they respected the idea that no single DLL or script becomes too large – it's effectively a suite of modules delivered together. *Why emulate:* If your domain is broad, consider breaking your module into a bundle of sub-modules. The Az design allows, for instance, the Networking team at MS to work on Az.Network independently from the Storage team on Az.Storage, as long as they both conform to common guidelines and depend on Az.Accounts. For internal distribution, you might similarly have, say, a "Core" module and then separate modules per domain (e.g., ADUserManagement vs. ExchangeManagement), which a roll-up can import all at once. The Az project also exemplifies rigorous testing – they have thousands of integration tests (though not open-sourced entirely) and a nightly build pipeline. They even provide an **AzPreview** module to let users try upcoming changes without affecting stable Az [76], highlighting an advanced versioning practice.

- **Pester** (yes, the testing framework itself): Pester as a module is interesting because it's both a consumer and provider of best practices. *Structure:* Pester is a single module but internally organized into multiple files for various components (assertion logic, test runner, etc.). It includes a **platyPS** generated help (about_Pester help file) and lots of tests (Pester has over 700 tests for itself!). One notable thing: Pester uses semantic versioning strictly and communicates breaking changes clearly (like the jump from 4.x to 5.x was significant and the maintainers provided migration guides). For module authors, following Pester's own evolution is instructive on how to manage a widely-used module (deprecations, aliasing old commands, etc. to maintain some backward compatibility). Pester's repository also shows how to integrate with CI: it runs on Azure Pipelines and GitHub Actions, and even its website (pester.dev) updates via CI. *Why emulate:* If your module becomes widely used, you may face similar challenges (ensuring old scripts don't break when people update your module). Pester's approach of documenting breaking changes and even providing a compatibility module for v4->v5 is something to learn from.

- **PSRule:** Already discussed in context of validation, **PSRule** is a Microsoft open-source module for policy compliance (by Bernie White and others). *Structure:* PSRule is actually a module that can act as a framework for other rule modules. The core PSRule module handles the engine; separate modules (like PSRule.Rules.Azure) package specific sets of rules. This separation is a best practice – keep the engine or library separate from content. PSRule's design encourages packaging rules into modules with a special tag so PSRule can auto-discover them [14]. The PSRule GitHub shows a sophisticated setup: they use **GitVersion** for versioning, **PlatyPS** to generate help markdown, **markdownlint** to keep documentation tidy [77] [78], and multi-platform CI (since PSRule runs on Linux in Azure Pipelines). They even provide VS Code extension integration for authoring rules [79] [13]. *Why emulate:* PSRule exemplifies writing a PowerShell module that isn't just a bunch of scripts, but a *framework* with DSL, config, and CI integration. It's heavily tested because reliability is key (nobody wants a

policy rule to fail intermittently). It's also built with distribution in mind – the authors note that rules can be shared via modules on the Gallery or private shares just like any module [80] [81]. If your project has a similar need (pluggability), look at how PSRule loads rule files, uses naming conventions (`*.Rule.ps1` files), and leverages the module manifest Tags and discovery.

Authoritative Resources: For further reading and as references during development: - *PowerShell Gallery Best Practices:* Microsoft's guide on module development and publication [82] [16] – covers documentation, tests, and the modules vs scripts argument. It's a checklist to ensure you haven't missed something before publishing. - *PowerShell Module Design on learn.microsoft.com:* Docs like "Writing a Windows PowerShell Module" and "How to Write a Module Manifest" give the official low-level details. These reiterate the benefits of modules for partitioning and reuse [83]. - *Community Blogs:* Many MVPs and experts blog about their module practices. For example, Mike F Robbins's blog post on script module design (monolithic vs drop-in functions) [84], or Adam Driscoll's posts on CI with PowerShell, etc. There's an active community on PowerShell.org and Reddit (/r/PowerShell) where patterns are discussed (often the question "how do I structure a large module?" gets answered with the principles we've described). - *GitHub Repos:* Don't hesitate to read source of great modules. Aside from those above, modules like **PSScriptAnalyzer** itself (how it's structured to plug into CI), **AWS Tools for PowerShell** (another huge set of modules per AWS service), or **dbatools** as mentioned. Microsoft's **Plaster** repo [85] and **PowerShellGet/PSResourceGet** repo are also educational to see how they structure tests and build (PSResourceGet, being written in C#, has a different structure but has a lot of PowerShell deployment logic). - *Books:* The PowerShell community has texts like "The PowerShell Module Builder's Guide" or even the classic "Month of Lunches" series touches on script modules. As of 2025, some new resources cover using AI assistance as well (which we'll get into next).

By looking at these examples and resources, one can avoid re-inventing wheels and adopt proven patterns. For instance, adopting dbatools' practice of one function per file and a shared internal functions file can save you from a lot of merge headaches. Or using PSFramework instead of writing your own logging. Standing on the shoulders of giants accelerates your modernization journey.

V. The Role of AI in Modern PowerShell Development

AI-powered coding assistants (like GitHub Copilot, ChatGPT, and others) have emerged as useful tools for PowerShell development. They can dramatically speed up writing boilerplate, suggest command sequences, and even generate initial test code. However, to use them in an *enterprise-safe* way, you must employ them within guardrails and best practices.

AI Assistance Benefits: AI coding agents are excellent at producing repetitive code patterns quickly. For example, Copilot can suggest the structure of a new function (parameters, basic cmdlet calls) after seeing a couple examples in your module. It can also help with things like writing Comment-Based Help sections or basic Pester tests. Teams have found that AI can reduce the tedium of writing all those parameter validation attributes or multiple similar functions (like generating 10 cmdlets for CRUD operations on similar resources).

Risks and Guardrails: AI suggestions are not guaranteed to be correct or optimal. They may contain subtle bugs, use outdated approaches, or not follow your internal standards. Therefore, treat AI as a junior pair-programmer whose work *must be reviewed*. Guardrails to implement: - **PSScriptAnalyzer and Custom Rules:** Always run any AI-generated code through PSScriptAnalyzer (which will catch obvious issues like missing `CmdletBinding`, unused variables, etc.). Many teams add custom PSScriptAnalyzer rules to

enforce internal standards; these will flag AI outputs that don't comply. - **Code Reviews:** Require human code review for any AI-generated code, just as you would for human-written code. During review, pay attention to security (AI might not sanitize inputs unless prompted), efficiency (it might suggest a less-efficient loop), and style consistency. - **Testing Requirements:** Insist that AI-generated code include tests (or at least that you write tests for it). If you prompt an AI to write a function, also prompt it to write accompanying Pester tests for that function. This not only helps validate the code, but also often exposes if the code doesn't actually do what was intended. For example, if Copilot writes a function and basic tests, running those tests could immediately highlight logical errors. - **Limit Production Direct Use:** Some orgs disallow directly deploying code with no human modification. In practice, an engineer might use AI to generate 80% of a new module, but then they will adapt it to the company's context and clean it up. Essentially, AI speeds up the grunt work but doesn't remove the engineering judgment.

GitHub Copilot in particular has features like **Custom Instructions** where you can tell it your style guidelines. Forward-looking teams even create Copilot "organization instructions" to enforce certain patterns (for instance, *always include* `Write-Verbose` *in catch blocks*, or *our functions must use ShouldProcess*). There's a blog by an MVP describing *"Copilot with enterprise standards via custom instructions"* [86] [87] – they created a repository of markdown instructions and prompts so that Copilot would produce code aligned to their security and logging requirements. For example, they have an instruction that any generated function must include logging of a correlation ID, input validation, no hardcoded creds, etc. [88] [89] . This is an emerging area, but it's relevant: you can *teach* your AI your framework's guardrails. If done well, Copilot could then generate a "pure Transformation function" or "idempotent State function" skeleton that already has the correct structure (like `SupportsShouldProcess` and try/catch blocks).

AI for Prompting Code Gen: Another safe way to use AI is via structured prompting templates. Instead of letting AI guess what you want, explicitly instruct it. For instance, you might maintain a set of prompt templates for generating code that adheres to your 5-category rules. Consider creating a prompt for each category. Below is an example **prompt template** to generate a **pure Transformation function**, ensuring the AI knows to avoid side effects and to include tests:

```
<task>Write a PowerShell advanced function that transforms input objects into
output objects without side effects.</task>
<requirements>
  - Use [CmdletBinding()] and strict parameter validation.
  - No file/network/registry/process mutations; no Write-Host/Write-Output
except pipeline return.
  - Input/Output schema documented in comment-based help (SYNOPSIS, DESCRIPTION,
PARAMETER, INPUTS, OUTPUTS, EXAMPLES).
  - Deterministic: same input → same output. Include 3 Pester unit tests.
</requirements>
<signature>Function name: Convert-CompanyXSomething (Verb-Noun, approved verb).
Return a typed object or hashtable.</signature>
<deliverables>
  1) Function code.
  2) Pester tests for happy-path, edge case, and error case.
```

```
  3) Brief rationale on purity and testability.
</deliverables>
```

This structured prompt explicitly asks the AI for an advanced function with CmdletBinding, with no side effects, with help comments, plus Pester tests. By providing such detailed instructions, you significantly increase the chance the AI's output meets your standards. The mention of examples ensures it includes usage examples in the help, which many AI outputs otherwise skip. And requesting tests is crucial: you get not just the code but a way to verify it. The AI's rationale can be used to quickly assess if it understood the concept of purity.

Similarly, you could have a prompt template for a **State Change function** focusing on idempotence and WhatIf, e.g.:

```
<task>Create an advanced function that ensures a target resource is in the
desired state.</task>
<requirements>
  - Supports -WhatIf and -Confirm; implements ShouldProcess.
  - Preconditions and postconditions documented; emits structured results.
  - Retry/backoff logic and error handling; compensating action on failure
(describe).
  - Include 2 Pester tests using mocks to prevent real mutations.
</requirements>
```

This would guide the AI to produce something with `if ($PSCmdlet.ShouldProcess(...)) { ... }`, maybe a try/catch with a retry loop, etc., plus tests that mock the actual change.

Governance and Policy: Enterprises using AI often set up policies like *"no code goes to production without passing all tests and analyzers"*. With AI in the mix, it's wise to also monitor for things like accidental inclusion of sensitive info. Copilot has filters, but one should ensure no secrets or internal server names ended up in prompts or outputs. Tools like Microsoft's **Copilot for Organizations** allow admin control and auditing of AI usage.

It's also worth noting that **AI can help in testing and analysis**, not just code writing. For instance, you might use AI to generate additional test cases or to perform a security review of your script. Microsoft is previewing features like "AI-assisted test generation" [90] – which could generate Pester tests by analyzing your code. Early trials suggest AI might catch some edge cases humans forgot. So teams can leverage that: after writing a module function, ask an AI "Write Pester tests for this function focusing on edge cases." Even if you don't use them verbatim, it might suggest a scenario you missed.

Conclusion on AI: Embrace AI as a productivity booster to follow our engineering framework – use it to quickly scaffold module components, to enforce patterns via prompt engineering, and to reduce toil (like writing documentation or repetitive parameter sets). But always keep it within the guardrails of automated checks and human oversight. When used wisely, AI becomes another tool in the DevOps toolkit, like an ever-improving snippet library that can adapt to your needs.

In practice, teams have reported that AI can handle the “boring 80%” of a task, letting the senior engineers focus on the complex 20%. For example, generating a dozen parameter sets for various AD user properties based on a class definition, or stubbing out all required functions of a module with correct comment-based help, which they then go fill in with real logic.

By integrating AI into your development workflow – but validating its output through tests and reviews – you accelerate development **without sacrificing quality**. The synergy of AI speed and human judgment, under the umbrella of a strong engineering framework, is a competitive advantage for modern PowerShell teams.

(The next sections include appendices with concrete examples and checklists that complement the topics above.)

Appendix A: Reference Directory Scaffolds

To solidify the concept of project layout, here are two reference scaffolds:

A.1 Minimal Module Structure – suitable for a small module or a starting template:

```
MyModule/
├── MyModule.psd1           # Module manifest with metadata and exports
├── MyModule.psm1           # Module file (imports private functions, exports
public)
├── src/
│   ├── Public/             # .ps1 files for each public function (to be
exported)
│   └── Private/            # .ps1 files for private helper functions (dot-
sourced)
└── tests/                  # Pester test files (*.Tests.ps1)
```

Notes: - The `Public` folder might contain files like `Get-Thing.ps1`, `Set-Thing.ps1`, etc. Each should have an `function Verb-Noun { [CmdletBinding()] param(...) ... }` inside. The module manifest’s `FunctionsToExport` can be either `@('Get-Thing','Set-Thing',...)` or `'*'` if you want to export all public functions automatically. - The `Private` folder contains helpers not meant for external use (e.g., `Convert-InternalFormat.ps1`). The `MyModule.psm1` would have lines like `. "$PSScriptRoot\src\Private\Convert-InternalFormat.ps1"` to include them. These are **not** exported in the manifest, so they remain internal. - The manifest (`MyModule.psd1`) also lists things like module version, author, license, and any RequiredModules. It also can include `PrivateData` for things like tags (e.g., if this module is a PSRule rules module, you’d put `Tags = @('PSRule-rules')` as seen earlier [91]). - Tests folder: typically, one test file per public function or per feature. e.g., `Get-Thing.Tests.ps1` with tests for `Get-Thing`. Pester will discover these automatically when you run `Invoke-Pester` (by default it finds `*.Tests.ps1`). Using a naming convention helps (and is required for Pester to auto-discover tests unless you specify script paths).

This minimal layout is easy to navigate and works for most cases. The separation of src and tests ensures you can easily run all tests or package the module without including test files.

A.2 Enterprise Module Structure (Expanded) – an example for a larger project or team-maintained module:

```
MyModule/
├── MyModule.psd1
├── MyModule.psm1
├── src/
│   ├── Public/
│   │    ├── Foo/
│   │    │    ├── Get-Foo.ps1
│   │    │    └── Set-Foo.ps1
│   │    └── Bar/
│   │         └── Invoke-Bar.ps1
│   └── Private/
│        ├── Helpers/
│        │    └── Convert-Stuff.ps1
│        └── InternalConfig.ps1
├── tests/
│   ├── Foo/
│   │    └── Get-Foo.Tests.ps1
│   ├── Bar/
│   │    └── Invoke-Bar.Tests.ps1
│   └── Integration/
│        └── FullScenario.Tests.ps1
├── build/
│   ├── psakefile.ps1       # psake build script defining tasks
│   └── azure-pipelines.yml # (if using Azure DevOps pipeline as code)
├── docs/
│   ├── about_MyModule.help.md  # conceptual help
│   └── ChangeLog.md
├── .github/
│   └── workflows/
│        └── ci.yml        # GitHub Actions pipeline definition
└── README.md
```

Notes: - In `src/Public`, an additional subfolder level (Foo, Bar) is used here to group related commands. This isn't strictly necessary, but in very large modules it can help manage hundreds of files. If you do this, you need to adjust how you dot-source them in the psm1 (e.g., recursively include files) or use a module loading approach that searches subdirectories. - The `build/` folder contains automation scripts. For instance, `psakefile.ps1` could define tasks like *Test*, *Analyze*, *Package*. The Azure DevOps YAML is included for reference so that the pipeline is version-controlled (some teams prefer to keep pipeline config with code). - `docs/` contains documentation that isn't code – e.g., conceptual about_ help. PlatyPS could

be used to generate this from code comments. The ChangeLog tracks changes per version (very important for communicating updates in enterprise). - The `.github/workflows` contains a GitHub Actions pipeline config (if using GitHub). Alternatively, if using Jenkins, you might have a `Jenkinsfile` in the repo root instead. - The root README.md gives an overview and usage examples for end users.

This structure covers all aspects: code (src), tests, build, docs, CI config. Not every project will have all these (for example, if you use Azure DevOps but not GitHub, you wouldn't need the .github folder; conversely, if not using Azure DevOps, you might not have azure-pipelines.yml). But it's illustrative of an enterprise project where everything is tracked.

Encapsulation & Compliance: In an enterprise, you often have rules like "no credentials in code" or "all public functions must have at least one Pester test." Organizing code and tests as above makes it easier to enforce those – for instance, you could write a script to ensure for each file in Public, a corresponding Tests file exists. Or run a grep across `src/` to ensure no hard-coded passwords, etc., possibly as part of the build.

Scaffolding Tools & Templates: Both Plaster and PSModuleDevelopment can be configured to generate structures like A.2. For example, a Plaster template can include conditions (like if the user chooses to include CI for GitHub, then create the .github folder). PSModuleDevelopment, as mentioned, allows you to have your own template that could produce exactly the above structure. Leveraging these ensures that every new module your team creates has uniform structure, making it easier for any engineer to jump between projects.

Appendix B: Example Pester Test Files by Category

Below are simplified examples of how Pester tests might look for different types of functions in our framework. These are illustrative – real tests would be more exhaustive – but they show the general approach:

B.1 Testing a Pure Transformation (Data Transformation category)

Let's say we have a function `Convert-WidgetToGadget` that takes a `[Widget]` object and returns a `[Gadget]` object (pure transformation, no side effects):

```
# File: tests/Convert-WidgetToGadget.Tests.ps1
$here = Split-Path -Parent $MyInvocation.MyCommand.Path
. "$here/../src/Public/Convert-WidgetToGadget.ps1"  # Import the function for
testing (if not in module)

Describe "Convert-WidgetToGadget" {
    It "converts a Widget to a Gadget with equivalent properties" {
        $widget = [PSCustomObject]@{ Name = "Test"; Size = 42 }
        $result = Convert-WidgetToGadget -InputObject $widget
        $result | Should -Not -Be $null
        $result.Name | Should -Be "Test"
        $result.Diameter | Should -Be 42  # assuming Gadget uses 'Diameter'
```

```
instead of 'Size'
    }

    It "throws an error if input is null" {
        { Convert-WidgetToGadget -InputObject $null } | Should -Throw
    }

    It "processes an array of Widgets (supports pipeline)" {
        $w1 = [PSCustomObject]@{ Name = "A"; Size = 1 }
        $w2 = [PSCustomObject]@{ Name = "B"; Size = 2 }
        $result = @($w1, $w2) | Convert-WidgetToGadget
        $result | Should -HaveCount 2
        $result[0].Name | Should -Be "A"
        $result[1].Size | Should -Be $null   # e.g., Gadget objects might not
have Size property
    }
}
```

Explanation: We import the function (for a unit test outside of module context). The tests then create dummy `Widget` objects and verify the output gadget has expected properties. We test normal conversion, an error case (null input), and pipeline support (if applicable). There's no external dependency here, no need for mocks because it's pure computation.

B.2 Testing a Data Acquisition Function (with external call)

Suppose `Get-WidgetData` calls a REST API using `Invoke-RestMethod`. We want to test it without calling the real API:

```
# File: tests/Get-WidgetData.Tests.ps1
. "$here/../src/Public/Get-WidgetData.ps1"

Describe "Get-WidgetData" {
    It "returns Widget objects for a valid query (API success)" {
        # Prepare a fake API response as would be returned by Invoke-RestMethod
        $fakeResponse = @{ widgets = @(
            @{ name="X"; size=10 },
            @{ name="Y"; size=20 }
        ) }
        # Mock Invoke-RestMethod to return fakeResponse instead of calling real
API
        Mock -CommandName Invoke-RestMethod -MockWith { return $fakeResponse } -
Verifiable

        $result = Get-WidgetData -Filter "size gt 5"
        # Validate that we processed the fakeResponse correctly
        $result | Should -All -BeOfType PSCustomObject
```

```
        (($result | Where-Object Name -EQ 'X').Size) | Should -Be 10

        # Ensure our function called Invoke-RestMethod once with correct
endpoint
        Assert-MockCalled Invoke-RestMethod -Times 1 -ParameterFilter { $URI -
match "/widgets" }
    }

    It "handles API exception and writes error" {
        # Simulate API throwing an exception (e.g., network error)
        Mock Invoke-RestMethod { throw "API down" } -Verifiable
        { Get-WidgetData -Filter "size lt 0" } | Should -Throw
        # Optionally, if our function catches and writes a warning instead:
        # (Get-WidgetData -Filter "size lt 0" -ErrorAction SilentlyContinue) |
Should -Be $null
        Assert-MockCalled Invoke-RestMethod -Times 1
    }
}
```

Explanation: We use `Mock Invoke-RestMethod` to intercept calls. In the first test, we set it to return a fake hashtable that mimics JSON data from an API. After calling our function, we verify that it returns objects as expected and that it indeed called Invoke-RestMethod once with an endpoint containing "/widgets" (just as an example filter). In the error test, we mock it to throw, then ensure our function surfaces an error (depending on design, we might expect it to throw or to catch and write a warning). We check that Invoke-RestMethod was called (so our function didn't, say, skip it entirely).

This isolates our function's logic (parsing the response, error handling) without depending on any real web service.

B.3 Testing a State Change Function (with WhatIf and Confirm)

Imagine `Set-WidgetSize` changes the size of a Widget in a database by calling an internal function `Update-WidgetInDB`. We want to test idempotence and ShouldProcess:

```
# File: tests/Set-WidgetSize.Tests.ps1
. "$here/../src/Public/Set-WidgetSize.ps1"
. "$here/../src/Private/Update-WidgetInDB.ps1"  # the internal function that
actually does DB update

Describe "Set-WidgetSize" {
    It "updates the widget when size is different" {
        # Prepare a dummy widget
        $widget = [PSCustomObject]@{ Id = 1; Size = 5 }
        # Mock the internal update function to simulate a DB update
        Mock Update-WidgetInDB -ParameterFilter { $Id -eq 1 -and $NewSize -eq
10 } -MockWith { return $true } -Verifiable
```

```
        $result = Set-WidgetSize -Widget $widget -Size 10 -Confirm:$false
        $result | Should -BeTrue  # assume function returns $true on success
        Assert-MockCalled Update-WidgetInDB -Times 1
    }

    It "does nothing if size is already the same (idempotent)" {
        $widget = [PSCustomObject]@{ Id = 2; Size = 7 }
        # We expect no call to Update-WidgetInDB because no change needed
        Mock Update-WidgetInDB -MockWith { $true }  # just in case, but we will
assert it's not called
        $result = Set-WidgetSize -Widget $widget -Size 7 -Confirm:$false
        $result | Should -BeFalse  # assume returns $false if no change
        Assert-MockCalled Update-WidgetInDB -Times 0
    }

    It "supports -WhatIf (does not call update)" {
        $widget = [PSCustomObject]@{ Id = 3; Size = 1 }
        Mock Update-WidgetInDB -MockWith { return $true }
        $output = Set-WidgetSize -Widget $widget -Size 99 -WhatIf
        # Check that output contains WhatIf message and function didn't throw
        $output | Should -Match "What if:"
        Assert-MockCalled Update-WidgetInDB -Times 0
    }

    It "prompts for confirmation when not forced" {
        $widget = [PSCustomObject]@{ Id = 4; Size = 2 }
        # Use -Confirm without -Confirm:$false to simulate user would be
prompted.
        # Pester can't easily simulate user input to confirm, but we can ensure
ShouldProcess logic is present:
        Mock Update-WidgetInDB -MockWith { $true }
        { Set-WidgetSize -Widget $widget -Size 3 -Confirm } | Should -Throw -
ErrorId "PipelineStopped"
        # Explanation: If -Confirm is given, PowerShell would prompt; in
unattended test, it'll throw PipelineStopped (as user did not confirm).
        Assert-MockCalled Update-WidgetInDB -Times 0
    }
}
```

Explanation: - In the first test, we ensure that if a widget needs updating, our function calls the internal DB update exactly once and returns true. We disable confirmation by `-Confirm:$false` to avoid prompt. We use `ParameterFilter` on the mock to ensure it only counts if called with expected args (Id=1, NewSize=10). - Second test ensures idempotence: if the desired size equals current size, the function should do nothing. We verify it *did not call* the update function and perhaps returned false or some indicator of "no action". - Third test checks `-WhatIf`: we expect no call to update and that a WhatIf message was produced. We didn't simulate capturing output here elegantly; one could use `$OutputPreference` or

temporarily override `$PSCmdlet.ShouldProcess()` via mocking – but simplest is to run it and ensure our mock wasn't invoked. We also inspect the output string for "What if". - Fourth test checks `-Confirm` behavior: When `-Confirm` is used, PowerShell will automatically prompt user. In Pester (non-interactive), if our function correctly calls `ShouldProcess("desc","target")`, PowerShell will throw a PipelineStopped exception because it can't confirm. We catch that and treat it as expected (hence the `Should -Throw -ErrorId "PipelineStopped"` – PipelineStopped is the typical error when a confirm prompt is not answered in automation). And again no update call should have happened. This indicates our function honored the confirmation (didn't proceed without confirmation).

These tests validate the **ShouldProcess implementation and idempotence logic** in the State Change function.

B.4 Testing a Configuration/Validation Function

Example: `Test-WidgetPolicy` ensures a widget meets certain criteria (size <= 100, name not empty, etc.) and returns $true/$false or a report. Tests would construct sample objects that violate or meet policy:

```
Describe "Test-WidgetPolicy" {
    It "passes compliant widget" {
        $widget = [PSCustomObject]@{ Name = "Valid"; Size = 50; Tags =
@("approved") }
        $result = Test-WidgetPolicy -Widget $widget
        $result | Should -BeTrue
        # Or if it returns an object with .Compliance property, check that
    }
    It "fails widget with no Name" {
        $widget = [PSCustomObject]@{ Name = ""; Size = 20 }
        $result = Test-WidgetPolicy -Widget $widget
        $result | Should -BeFalse
        # Perhaps also test that a specific warning or error was produced
        ($Global:Error[0].Exception.Message) | Should -Match "Name is required"
    }
}
```

This is straightforward – just ensuring logic catches the problem. If using PSRule, one might actually run Invoke-PSRule in tests to see the outcome.

B.5 Testing an Orchestrator

For an orchestrator `Invoke-WidgetDeployment` that calls multiple steps, we rely on mocking those steps:

```
Describe "Invoke-WidgetDeployment (Orchestrator)" {
    It "calls sub-functions in order and succeeds" {
        Mock Get-WidgetData -MockWith { @([PSCustomObject]@{Id=1; Name="W1";
```

```
Size=5}) } -Verifiable -Sequence
        Mock Set-WidgetSize  -MockWith { $true } -Verifiable -Sequence
        Mock Test-WidgetPolicy -MockWith { $true } -Verifiable -Sequence

        $ok = Invoke-WidgetDeployment -WidgetId 1 -NewSize 5 -Confirm:$false
        $ok | Should -BeTrue
        # Verify each was called exactly once
        Assert-MockCalled Get-WidgetData -Times 1 -ParameterFilter { $WidgetId -
eq 1 }
        Assert-MockCalled Set-WidgetSize -Times 1 -ParameterFilter { $widget.Id
-eq 1 -and $Size -eq 5 }
        Assert-MockCalled Test-WidgetPolicy -Times 1
    }

    It "rolls back if Set-WidgetSize fails" {
        Mock Get-WidgetData -MockWith { @([PSCustomObject]@{Id=2; Name="W2";
Size=5}) }
        Mock Set-WidgetSize -MockWith { throw "DB failure" } -Verifiable
        Mock Undo-WidgetChanges -MockWith { "rolled back" } -Verifiable

        { Invoke-WidgetDeployment -WidgetId 2 -NewSize 99 -Confirm:$false } |
Should -Throw "DB failure"
        Assert-MockCalled Undo-WidgetChanges -Times 1 -ParameterFilter {
$WidgetId -eq 2 }
    }
}
```

Explanation: - In the first test, we mock each sub-step (Get, Set, Test). The `-Sequence` and the order of Mock definitions enforce they are used in order (though Pester's Sequence is advanced usage; here mostly we just care they were called once). We then assert they were each called with expected parameters. We assume the orchestrator returns $true on success. - In the second test, we simulate a failure in the middle (Set-WidgetSize throws). We expect our orchestrator catches it and calls an `Undo-WidgetChanges` (a hypothetical cleanup function). We verify the undo was called once. We also ensure the orchestrator bubbled an error (or maybe it would return a failure status; here we assumed it throws or allows error to propagate).

These tests verify that the orchestration handles flows correctly (calls things, and does rollback on error).

Conclusion: Each category's tests have a different flavor: - Pure functions: direct input/output assertions. - External calls: use `Mock` to isolate environment. - State changes: test WhatIf/Confirm and idempotence via control of ShouldProcess (implicitly through PipelineStopped behavior or explicit checks). - Orchestrators: heavy use of `Assert-MockCalled` to ensure sequence.

By following these patterns, you achieve high confidence in each part of the system. When all parts are tested in isolation, the integrated system is much less likely to surprise you – and if it does, you can narrow down which part's assumption was wrong.

Appendix C: Sample CI/CD Pipeline Configurations

This appendix provides snippets for CI pipelines – one for **GitHub Actions** and one for **Azure DevOps** – focusing on running tests and publishing modules. (These are illustrative; adapt paths and keys to your environment.)

C.1 GitHub Actions – CI Pipeline (YAML)

```
# .github/workflows/ci.yml
name: Build, Test and Publish

on:
  push:
    branches: [ main ]

jobs:
  build-test-publish:
    runs-on: ubuntu-latest  # Use a Windows runner if Windows-specific code
    steps:
      - name: Checkout repository
        uses: actions/checkout@v3

      - name: Setup PowerShell
        uses: PowerShell/PowerShell@v1  # Installs latest PowerShell on Ubuntu

      - name: Install PSScriptAnalyzer
        run: pwsh -Command "Install-Module PSScriptAnalyzer -Scope CurrentUser -
Force"

      - name: Static Code Analysis
        run: pwsh -Command "Invoke-ScriptAnalyzer -Path ./MyModule -Recurse -
Severity Warning,Error -OutPath SAScanResults.xml; if((Get-Content
SAScanResults.xml) -match 'Severity=\"Error\"') { write-host 'Static analysis
found errors'; exit 1 }"
      # The above runs analyzer and if any Error severity issues are found,
exits with failure.

      - name: Install Pester
        run: pwsh -Command "Install-Module Pester -Scope CurrentUser -Force"

      - name: Run Tests
        run: pwsh -Command "Invoke-Pester -Path .\tests -OutputFormat NUnitXml -
OutputFile TestResults.xml -CI:$true"
      # -CI switch in Pester 5 turns off colors etc. for CI systems.

      - name: Publish Test Results
```

```
        uses: actions/upload-artifact@v3
        if: always()   # always upload results even if tests fail
        with:
          name: PesterResults
          path: TestResults.xml

      - name: Publish Module to Internal Repository
        if: ${{ success() && github.ref == 'refs/heads/main' }}
        env:
          FEED_URL: ${{ secrets.MY_FEED_URL }}           # e.g., NuGet feed URL
          FEED_API_KEY: ${{ secrets.MY_FEED_API_KEY }}   # e.g., PAT or API key
for feed
        run: |
          pwsh -Command "Install-Module Microsoft.PowerShell.PSResourceGet -
Scope CurrentUser -Force"
          pwsh -Command "$cred = [PSCredential]::new('anyuser', (ConvertTo-
SecureString '${{ secrets.FEED_API_KEY }}' -AsPlainText -Force)); Register-
PSResourceRepository -Name MyFeed -Uri $Env:FEED_URL -Credential $cred -Trusted
-Priority 0"
          pwsh -Command "Publish-PSResource -Path ./MyModule -Repository MyFeed
-NuGetApiKey 'FeedKey'"
```

Explanation: This GH Actions pipeline does the following: - Checks out code. - Sets up PowerShell on the runner. - Installs PSScriptAnalyzer and runs it on `./MyModule` directory (assuming module code is in MyModule folder). If any "Error" issues are found, it fails the job (exit 1). Warnings are allowed but one could decide to treat warnings as fail too. - Installs Pester (though on ubuntu-latest, Pester 5 is likely preinstalled, but explicit is fine). - Runs Pester on the `tests` directory, outputting results to an NUnit XML file. - Always uploads the XML as an artifact (so you can download the test results from the Actions run, or use another action to convert them to GitHub test annotations). - If everything succeeded and we are on the main branch, it then publishes the module: - It installs PSResourceGet (in case not present). - It registers an internal repository named MyFeed using a URL and an API key stored in secrets (secrets are encrypted values you configure in GitHub; here maybe FEED_URL is something like `https://nuget.pkg.github.com/<org>/index.json` for GitHub Packages, or an Azure Artifacts URL). - It then calls `Publish-PSResource -Path ./MyModule` to pack and push the module. We pass a dummy `-NuGetApiKey 'FeedKey'` because some NuGet feeds require an API key parameter even if using PAT via credentials (for GitHub, `NuGetApiKey` would be the PAT; for Azure Artifacts, it can be any string if using credential auth – in this script we added the PAT as a credential which PSResourceGet will use, and just passed a filler string as ApiKey because the cmdlet requires it). The specifics vary by feed type.

This pipeline demonstrates **continuous integration** (lint + test) and **continuous delivery** (auto-publish on main branch). For a production scenario, you might want to include a manual approval or version bump check before publishing to a gallery.

C.2 Azure DevOps – Pipeline YAML for Module

If using Azure DevOps (or Azure DevOps Server), a pipeline YAML might look like:

```yaml
# azure-pipelines.yml
trigger:
- main

pool:
  vmImage: 'windows-latest'  # Using Windows runner for simplicity

steps:
- task: PowerShell@2
  displayName: Install PSScriptAnalyzer and Pester
  inputs:
    targetType: 'Inline'
    script: |
      Install-Module PSScriptAnalyzer -Scope CurrentUser -Force
      Install-Module Pester -Scope CurrentUser -Force

- task: PowerShell@2
  displayName: Run Static Analysis
  inputs:
    targetType: 'Inline'
    script: |
      $results = Invoke-ScriptAnalyzer -Path $(System.DefaultWorkingDirectory)
\MyModule -Recurse -Severity Warning,Error
      if ($results.Where({ $_.Severity -eq 'Error' })) {
        Write-Host "PSScriptAnalyzer found errors"
        $results | ConvertTo-Json | Out-File SAScanResults.json
        Exit 1
      }
      # (Could also publish SAScanResults as build artifact if needed)
  failOnStderr: false

- task: PowerShell@2
  displayName: Run Pester Tests
  inputs:
    targetType: 'Inline'
    script: |
      Invoke-Pester -Path $(System.DefaultWorkingDirectory)\tests -OutputFormat
NUnitXml -OutputFile $(System.DefaultWorkingDirectory)\TestResults.xml
  continueOnError: true

- task: PublishTestResults@2
  displayName: Publish Test Results
  inputs:
    testResultsFiles: '$(System.DefaultWorkingDirectory)\TestResults.xml'
    testRunner: 'NUnit'
    failTaskOnFailedTests: true
```

```
- task: PowerShell@2
  displayName: Publish Module to Azure Artifacts
  condition: succeeded()
  env:
    SYSTEM_ACCESSTOKEN: $(System.AccessToken)   # Azure DevOps built-in token
for auth
  inputs:
    targetType: 'Inline'
    script: |
      # Install PSResourceGet if not already

Install-Module -Name Microsoft.PowerShell.PSResourceGet -Scope CurrentUser -
Force
      Register-PSResourceRepository -Name "OrgFeed" -URI "https://
pkgs.dev.azure.com/<Org>/<Project>/_packaging/<Feed>/nuget/v3/index.json" -
AuthAccessToken $env:SYSTEM_ACCESSTOKEN -Trusted
      Publish-PSResource -Path $(System.DefaultWorkingDirectory)\MyModule -
Repository "OrgFeed" -NuGetApiKey "AzureDevOps"
```

Explanation: - Uses Windows runner (which has Windows PowerShell and PowerShell Core). - Installs PSScriptAnalyzer and Pester. - Runs ScriptAnalyzer; if any errors, logs and exits (failing the pipeline). Could also output to JSON for record. - Runs Pester and produces an NUnit XML. - Publishes that result so Azure DevOps shows test outcomes in Tests tab. `failTaskOnFailedTests: true` will mark pipeline red if any tests failed. - Finally, if all succeeded, uses `Publish-PSResource` to publish to an Azure Artifacts feed. Here, we leverage the `System.AccessToken` (Azure DevOps provides this token to authenticate the pipeline to its own feeds). PSResourceGet supports using ADO PATs (the System.AccessToken is essentially a PAT) through `-AuthAccessToken` parameter on Register-PSResourceRepository (this is hypothetical usage as of writing – if not supported directly, an alternative is to use `nuget push` CLI or the old Publish-Module with an Azure Artifacts credential provider, but PSResourceGet 1.1 does support ADO feeds as per docs). We mark repository as trusted and then call Publish. The `-NuGetApiKey "AzureDevOps"` is a dummy required by the API – Azure Artifacts doesn't require an actual API key string when using a token, but the cmdlet currently requires some string.

This pipeline will automatically run on pushes to main, run tests, and publish the module to the named feed if tests pass. In Azure DevOps, one might separate this into build and release pipelines; here shown as one for simplicity. Azure Pipelines could also integrate things like code coverage (using Coverlet for C# or Pester's CodeCoverage for PS, though Pester's code coverage output can be converted to formats Azure DevOps understands with some work).

C.3 Jenkins (declarative pipeline snippet) for completeness:

```
pipeline {
    agent any
    stages {
        stage('Build') {
            steps {
```

```
                powershell 'Install-Module PSScriptAnalyzer -Scope CurrentUser -
Force; Install-Module Pester -Scope CurrentUser -Force'
            }
        }
        stage('Analyze') {
            steps {
                powershell '''
                  $errs = Invoke-ScriptAnalyzer -Path .\\MyModule -Recurse -
Severity Error
                  if ($errs) {
                    $errs | ConvertTo-Html | Out-File AnalysisReport.html
                    Write-Error "Static Analysis failed with $($errs.Count)
errors."
                  }
                '''
            }
        }
        stage('Test') {
            steps {
                powershell '''
                  Invoke-Pester -OutputFile TestResults.xml -OutputFormat
NUnitXml -ErrorAction Continue
                '''
            }
        }
    }
    post {
        always {
            junit 'TestResults.xml'   // Publish test results
            archiveArtifacts artifacts: 'AnalysisReport.html',
onlyIfSuccessful: false
        }
        failure {
            mail to: 'dev-team@example.com',
                 subject: "Build ${env.BUILD_NUMBER} Failed",
                 body: "Check Jenkins for details."
        }
        success {
            powershell '''
              # On success, publish module to internal feed
              Import-Module Microsoft.PowerShell.PSResourceGet -ErrorAction Stop
              Register-PSResourceRepository -Name CorpFeed -Uri "https://
nuget.internal.corp/nuget/v3/index.json" -Credential (Get-Credential) -Trusted
              Publish-PSResource -Path .\\MyModule -Repository CorpFeed -
NuGetApiKey 'anything'
            '''
        }
```

```
    }
}
```

In this Jenkinsfile: - We use PowerShell steps to do analysis and testing. - We always publish test results (via JUnit plugin) and archive the analysis report (if any). - On success, we run a post step to publish the module (here using an interactive `Get-Credential` is not ideal for CI; in practice, one would use a stored credential or API key from Jenkins credentials store, injected as environment variables and then create PSCredential object in script). - On failure, notify via email (example).

Jenkins gives full flexibility, but more manual setup (ensuring the agent has PowerShell and modules, etc.). Many have migrated to cloud CI like GitHub Actions or Azure DevOps for easier maintenance, but Jenkins is still prevalent on-prem.

Key Takeaway: Regardless of platform, the CI pipeline should: - Run PSScriptAnalyzer (quality gate) [62]. - Run Pester tests and report results [31]. - Fail on any issues (preventing bad code from merging or releasing). - If all good, automate the packaging/publishing of the module (so developers don't do it manually, reducing errors).

By implementing such pipelines, you enforce that the engineering framework is followed on every code change. It's like having an automated supervisor: if someone introduces a side-effect in a supposed pure function, a test should catch it; if someone forgets to add `SupportsShouldProcess`, perhaps a PSScriptAnalyzer rule (like PSUseShouldProcess) will flag it. The pipeline then stops the code from proceeding until it's fixed. This ensures long-term codebase health and frees developers from constantly doing manual checks.

Appendix D: Internal Gallery & Versioning Checklist

When moving to a modular, engineered approach, managing distribution and versions of modules internally is crucial. Use this checklist to ensure smooth operations:

- **Set up an Internal Repository:** Choose a host for your modules:
- Azure Artifacts feed (recommended if you use Azure DevOps) – supports scoped feeds and permissions.
- GitHub Packages – if your code is on GitHub and you want to distribute via GitHub's package registry.
- File share or internal NuGet server – for completely offline or simple setups.
- Azure Container Registry – can serve as a PowerShell repository (from PSResourceGet 1.1 onward) [41].

Make sure all developers have **credentials** and instructions to register the repository (e.g., run a one-time `Register-PSResourceRepository` command) and that build agents/CI have access (use secret variables for API keys or tokens).

- **Enforce Semantic Versioning:** Every module should have a clear version in its manifest. Follow semver:
- Bump **Major** version for breaking changes (incompatible API changes, removed functions, changed parameter behavior).

- Bump **Minor** for new features (new functions/cmdlets or significant enhancements).
- Bump **Patch** for bug fixes or minor tweaks that are fully backward-compatible.

Document these changes in a **CHANGELOG** file or Release Notes. This helps consumers know whether they can safely update. The PS Gallery or other repositories will show the version, but internal users appreciate a summary of changes.

- **Include Module Metadata:** In the manifest's PrivateData/PSData, add:
- **Tags** (useful for discovery, e.g., tag modules with company name or purpose; required if distributing PSRule rules: tag `PSRule-rules` [91]).
- **ProjectURI / IconURI** if applicable (internal docs site or Git repo link).
- **LicenseURI** if needed (even internally, nice to indicate if proprietary).

- This metadata helps later if you have an internal gallery website or to keep track of module purpose.

- **Automated Build & Release Pipeline:** (As demonstrated in Appendix C)

- CI should create the module package and either automatically publish it (for continuous delivery) or make it available for a release process.
- Ensure the pipeline uses the updated module version from the manifest. You might automate version bumping (some teams have a script to increment version number based on Git commit messages or manual trigger).

- Use pipeline variables for sensitive info like API keys, never hard-code in scripts.

- **Signing (if required by policy):** Some enterprises mandate code signing for scripts/modules. If so, set up the infrastructure:

- Have a code signing certificate (from internal CA or public, depending on policy).
- Use PowerShell's `Set-AuthenticodeSignature` in the build pipeline to sign the `.psm1`, `.psd1`, and any script files. Or sign the nupkg after packaging.
- Ensure all agents have access to the cert (or use an HSM or Azure KeyVault signing task).
- If you cannot sign in CI, at least sign the published module package manually and redistribute. (But automated is better.)

- Signing allows execution policy to be AllSigned, increasing security.

- **Module Dependency Management:** If modules depend on each other (e.g., a Core module that others need):

- Publish the dependencies first.
- In the dependent module's manifest, specify `RequiredModules` with minimum versions [45].
- When publishing to an internal feed, ensure that feed has all required modules (or the clients have those modules from other sources). Possibly maintain a **central feed** with all approved modules (both your internal and any third-party ones you allow).

- Test installation on a clean system to verify that `Install-PSResource MyModule` pulls all it needs (maybe in a CI job, spin up a fresh container or VM, run install from feed).

- **Retention and Cleanup:** Over time, many versions might accumulate.

- Decide how many old versions to keep in the feed (especially if using Azure Artifacts with storage limits). Perhaps keep the last N versions or those from last Y years.
- However, if older scripts in production rely on old versions, coordinate deprecation carefully. Ideally, update all uses to current versions, then you can hide or remove old ones.

- Azure Artifacts allows deprecating or deleting old packages. Do so with caution (perhaps automate a warning – e.g., mark as deprecated in feed, give teams a month, then remove).

- **Documentation and Discovery:** Make it easy for team members to find and use modules:

- Maintain an internal catalog (a simple SharePoint or DevOps wiki page listing available modules with description and latest version).
- Encourage use of `Find-PSResource -Repository MyFeed` to search for modules by name or tag.

- Possibly create a “metapackage” module that lists all your modules as RequiredModules – not common, but some orgs do a "Company.AllModules" that just depends on others for one-step install of everything.

- **Rollout Strategy:** When a new module version is published:

- If backward-compatible, users can update at will. If breaking, communicate in advance. Possibly use feature flags or compatibility switches if needed for a transition.
- For critical automation, test the new module in a staging environment. Some pipelines might explicitly install version X to ensure consistency. As a practice, you might version-pin modules in production scripts to avoid surprise breaks, and then update those pins during planned maintenance.

- Leverage **PSResourceRepository priorities**: you can register multiple repositories with priority. E.g., an "InternalStable" feed with vetted releases priority 0, and "InternalBeta" feed with priority 1. This way, by default `Install-PSResource` pulls stable. If a user/developer wants a beta, they explicitly specify `-Repository InternalBeta`. This model can enable phased rollouts.

- **Monitoring and Feedback:** Once modules are widely used, gather feedback:

- Monitor the feed for download counts or who’s consuming what (Azure Artifacts has some stats; otherwise, logs or a manual survey).
- Have a channel (Teams/Slack or email) for users to report issues or requests. Treat the modules like products.
- If using PSFramework or your own logging, modules could emit usage telemetry to a central log (respect privacy and use only within org!). For instance, log an info whenever a module function is invoked, to know what's heavily used.

- Schedule periodic review of modules – e.g., quarterly check if any module can be consolidated, or if any new scripts should be converted into modules.

- **Backup the Repository:** If using Azure DevOps or GitHub cloud feeds, it's generally reliable. But if using an on-prem NuGet or file share, ensure it's backed up. The feed becomes a critical asset (it is effectively your internal PowerShell Gallery).

- It could be as simple as keeping the nupkg files in a secured location (since modules are code, they should also be in source control, but the feed with all versions is worth preserving too).

- **Security Scanning:** Consider running security scans on published modules:

- This could be part of CI (PowerShell Code Analysis via PSScriptAnalyzer catches some security issues).
- There are also tools like PowerShell ScriptAnalyzer Security rules or even running the module through Defender for DevOps (which can scan scripts for secrets).

- If modules call external systems or APIs, double-check that handling of credentials and data is secure (no storing plain text creds, etc.). Possibly integrate secret scanning in pipeline (GitHub and Azure DevOps have secret scanning features).

- **Decommissioning Plan:** If a module becomes obsolete or replaced by another, have a strategy:

- Mark it as deprecated (maybe add a warning in its functions using `[Obsolete()]` attribute in code or simply in documentation).
- Communicate to users which module or method replaces it.
- Keep the old version for a grace period then remove from feed (to prevent new use).
- This prevents "module sprawl" where old modules linger forever.

Using this checklist, organizations can maintain a healthy ecosystem of internal modules: All engineers know where to get modules (internal gallery), trust that modules are versioned and tested, and the platform (CI + repository) ensures distribution is consistent. It transforms script-sharing from copy-paste chaos into a governed process similar to how developers share libraries in traditional software development [82] [83] – reinforcing the motto: *treat your PowerShell scripts as reusable software components*.

---

[1] [2] [3] [5] [6] [7] [20] [25] [27] Benefits to breaking down script into functions/modules? : r/PowerShell
https://www.reddit.com/r/PowerShell/comments/1j3uiyk/benefits_to_breaking_down_script_into/

[4] What are the advantages/disadvantages of monolithic PHP coding ...
https://stackoverflow.com/questions/1562933/what-are-the-advantages-disadvantages-of-monolithic-php-coding-versus-small-spec

[8] [9] [10] [11] [26] Everything you wanted to know about ShouldProcess - PowerShell | Microsoft Learn
https://learn.microsoft.com/en-us/powershell/scripting/learn/deep-dives/everything-about-shouldprocess?view=powershell-7.5

[12] PSRule - Lessons in improving your Azure Infrastructure as Code ...
https://blog.makerx.com.au/psrule-lessons-in-improving-your-infrastructure-as-code-testing/

[13] [79] [80] [81] Features - PSRule
https://microsoft.github.io/PSRule/v1/features/

14 15 34 91 Packaging rules in a module - PSRule
https://microsoft.github.io/PSRule/v3/authoring/packaging-rules/

16 32 33 45 62 82 PowerShell Gallery Publishing Guidelines and Best Practices - PowerShell | Microsoft Learn
https://learn.microsoft.com/en-us/powershell/gallery/concepts/publishing-guidelines?view=powershellget-3.x

17 18 19 65 66 67 68 guidelines – dbatools
https://dbatools.io/join-us/guidelines/

21 22 64 Building and testing PowerShell - GitHub Docs
https://docs.github.com/en/actions/tutorials/build-and-test-code/powershell

23 24 29 30 Quick Start | Pester
https://pester.dev/docs/v4/quick-start

28 46 Visual Studio Code | Pester
https://pester.dev/docs/usage/vscode

31 Pester - The ubiquitous test and mock framework for PowerShell | Pester
https://pester.dev/

35 36 37 38 39 40 43 44 63 Use an Azure Artifacts feed as a private PowerShell repository
https://learn.microsoft.com/en-us/azure/devops/artifacts/tutorials/private-powershell-library?view=azure-devops

41 42 Use ACR repositories with PSResourceGet - PowerShell | Microsoft Learn
https://learn.microsoft.com/en-us/powershell/gallery/powershellget/how-to/use-acr-repository?view=powershellget-3.x

47 48 85 GitHub - PowerShellOrg/Plaster: Plaster is a template-based file and project generator written in PowerShell.
https://github.com/PowerShellOrg/Plaster

49 50 51 52 53 54 55 56 When Good Enough Isn't Good Enough: Customizing Module Templates - pr0mpt
https://www.pr0mpt.com/2025-07-06-when-good-enough-isnt-good-enough---customizing-module-templates/

57 58 59 60 Quick Start | psake
https://psake.netlify.app/docs/intro

61 nightroman/Invoke-Build: Build Automation in PowerShell - GitHub
https://github.com/nightroman/Invoke-Build

69 70 71 72 GitHub - PowershellFrameworkCollective/psframework: A module that provides tools for other modules and scripts
https://github.com/PowershellFrameworkCollective/psframework

73 74 75 76 Optimize the installation of Azure PowerShell | Microsoft Learn
https://learn.microsoft.com/en-us/powershell/azure/install-azps-optimized?view=azps-14.4.0

77 78 GitHub - microsoft/PSRule: Validate infrastructure as code (IaC) and objects using PowerShell rules.
https://github.com/microsoft/PSRule

83 Writing a Windows PowerShell Module - PowerShell | Microsoft Learn
https://learn.microsoft.com/en-us/powershell/scripting/developer/module/writing-a-windows-powershell-module?view=powershell-7.5

84 PowerShell Script Module Design Philosophy - mikefrobbins.com
https://mikefrobbins.com/2018/09/21/powershell-script-module-design-philosophy/

86 87 88 89 Transforming PowerShell Development with GitHub Copilot: Enterprise Standards and Custom Instructions
https://www.techbyjeff.net/transforming-powershell-development-with-github-copilot-enterprise-standards-and-custom-instructions/

90 AI-assisted test authoring with GitHub Copilot (preview) - Power ...
https://learn.microsoft.com/en-us/power-platform/test-engine/ai-authoring