PowerShell in CI/CD Pipelines

A Practical Tour of Deployment Options

Blake Cherry

: in/rblakecherry

Danny Stutz

X: @danny_stutz

Command Prompt

```
C:\Users\dstutz>whoami
Danny Stutz (@danny_stutz)
Computer nerd, M365 consulting and automation leader at West Monroe

Experience:
6 years working as a M365/Entra/Azure/AWS technical lead

Credentials:
West Monroe PowerShell Interest Group lead.
PowerShell + DevOps Global Summit alum

**********************************************************************************************
Ask Me About:

- My mechanical keyboard! (Keychron K2, Boba U4's)
- Migration tool automation with PowerShell (ShareGate, MigrationWiz, azcopy, etc...)
- Threat hunting, M365/Entra/Azure/AWS security
- Executing corporate divestiture & integration migration work (where I spend the majority of my consulting time)
- My music taste (send me your favorite tunes on Discord or Slack!)

**********************************************************************************************
Contact:

- x.com/danny_stutz
- linkedin.com/in/daniel-stutz
- github.com/danstutz
- dstutz ATSIGN westmonroe.com


C:\Users\dstutz>
```

PowerShell 7 (x64)

```
C:\Users\bcherry>whoami
Blake Cherry (@blakelishly)
Cybersecurity & Enterprise Technology Principal at West Monroe

Experience:
4 yrs. in consulting, focused on supporting divestitures and integrations
through automation and cloud/on-premises infrastructure solutions.

Credentials:
Lead for West Monroe greenfield environment build automation tooling. West Monroe Infrastructure Automation Interest Gro
up lead.

**********************************************************************************************************
Ask Me About:

- PowerShell and Infrastructure as Code Automation
- My homelab and self-hosted services
- Azure Cloud
- DevOps and pipeline automation
- Purdue Boilermakers football and basketball

**********************************************************************************************************
Contact:

- DISCORD/blakelishly
- linkedin.com/in/rblakecherry
- github.com/blakelishly
- bcherry@westmonroe.com

C:\Users\bcherry>whoami
```

Thank you Sponsors

Key Learning Objectives

- **Understand When & Why:** Learn the scenarios when PowerShell can be used in CI/CD pipelines to enhance your workflows.
- **Explore Pipeline Deployment Methods:** Discover the various approaches for running PowerShell in pipelines—including the use of containers and custom Dockerfiles—and understand their associated considerations.
- **Evaluate CI/CD Platforms:** Gain insights into the capabilities of different CI/CD tools for running PowerShell and how to select the best solution for your specific needs.

Agenda

- An Overview of PowerShell & CI/CD Pipelines
- Anatomy of a CI/CD Pipeline
- Platform Comparison & Tooling Insights
- Practical Use Cases & Integrations
- Demo
- Conclusion

A Tale as Old as Time

The Scenario:

You've crafted a PowerShell script **`Set-Everything.ps1`** to apply critical security configurations to servers.

A Tale as Old as Time

- **However**, operationalizing your script has been a nightmare:

A Tale as Old as Time

- **However**, operationalizing your script has been a nightmare:
  - **Manual Triggers & Targeting**: Your team RDPs to privileged workstations to execute the script across server lists. The approach is tedious, error-prone, and its easy to miss a target.

A Tale as Old as Time

- **However**, operationalizing your script has been a nightmare:
  - **Manual Triggers & Targeting**: Your team RDPs to privileged workstations to execute the script across server lists. The approach is tedious, error-prone, and its easy to miss a target.
  - **The "Works on My Machine" Lottery**: Your PowerShell environment rarely matches your coworker's leading to issues and constant debugging.

A Tale as Old as Time

- **However**, operationalizing your script has been a nightmare:
  - **Manual Triggers & Targeting**: Your team RDPs to privileged workstations to execute the script across server lists. The approach is tedious, error-prone, and its easy to miss a target.
  - **The "Works on My Machine" Lottery**: Your PowerShell environment rarely matches your coworker's leading to issues and constant debugging.
  - **Version Control Chaos**: Scripts live on network shares, maybe SharePoint? Edits happen directly. Rollbacks mean finding the "old version" and manually running it again.

A Tale as Old as Time

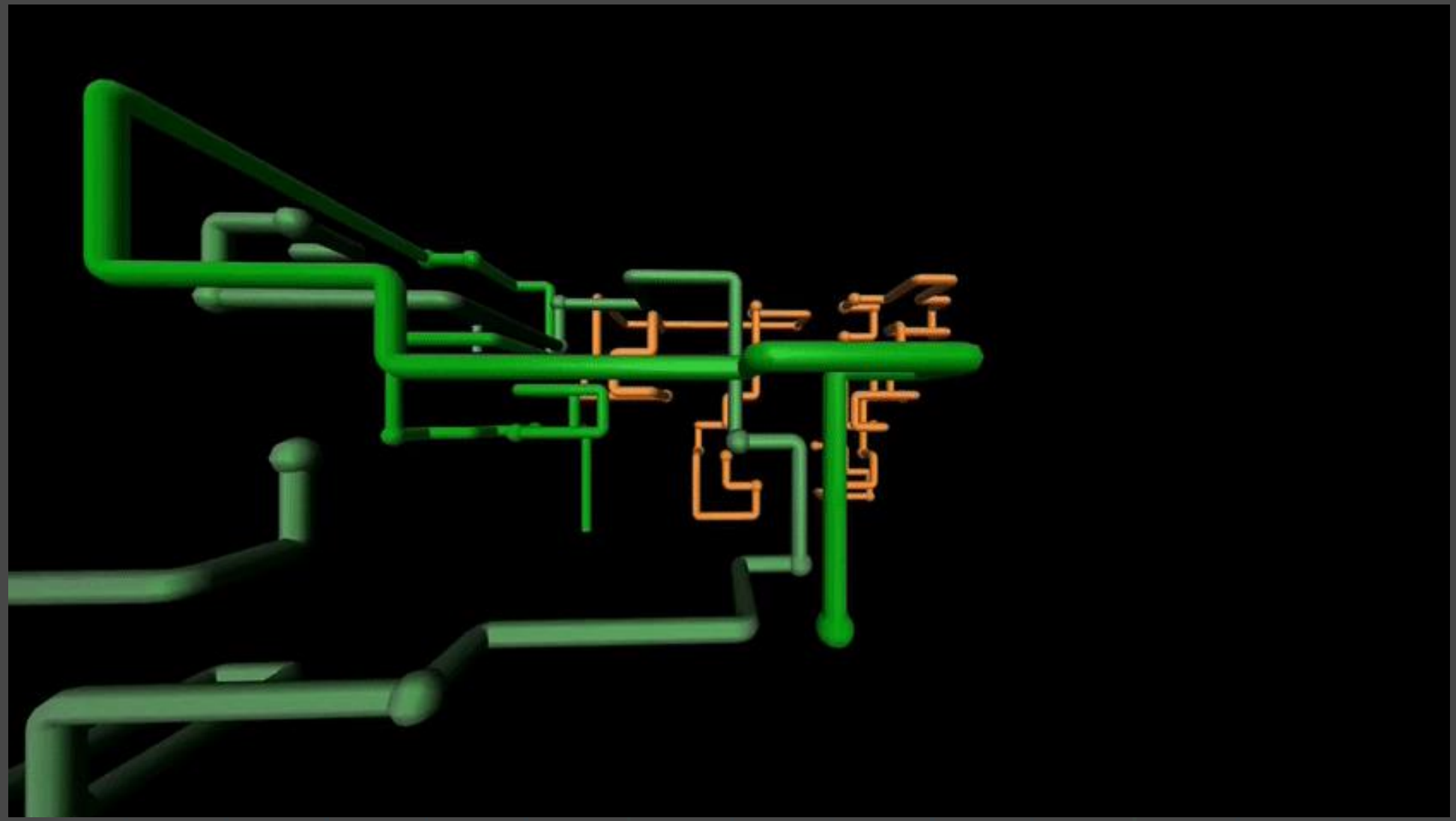
- **However**, operationalizing your script has been a nightmare:
  - **Manual Triggers & Targeting**: Your team RDPs to privileged workstations to execute the script across server lists. The approach is tedious, error-prone, and its easy to miss a target.
  - **The "Works on My Machine" Lottery**: Your PowerShell environment rarely matches your coworker's leading to issues and constant debugging.
  - **Version Control Chaos**: Scripts live on network shares, maybe SharePoint? Edits happen directly. Rollbacks mean finding the "old version" and manually running it again.
  - **Testing = High Stakes Gambling**: Changes often get their first real test in pre-prod (or worse, prod!). Finding bugs after execution requires manual fixes and messy re-runs.

A Tale as Old as Time

- **However**, operationalizing your script has been a nightmare:
  - **Manual Triggers & Targeting**: Your team RDPs to privileged workstations to execute the script across server lists. The approach is tedious, error-prone, and its easy to miss a target.
  - **The "Works on My Machine" Lottery**: Your PowerShell environment rarely matches your coworker's leading to issues and constant debugging.
  - **Version Control Chaos**: Scripts live on network shares, maybe SharePoint? Edits happen directly. Rollbacks mean finding the "old version" and manually running it again.
  - **Testing = High Stakes Gambling**: Changes often get their first real test in pre-prod (or worse, prod!). Finding bugs after execution requires manual fixes and messy re-runs.
  - **Inconsistent Application**: Your team member keeps forgetting to execute the script with the "-Force" parameter, causing issues on some target nodes.

A Tale as Old as Time

The Solution: Pipelines



The Solution: CI/CD Pipelines

- CI/CD Pipeline – not a | character
- What is a CI/CD Pipeline?
  - A continuous integration and continuous deployment (CI/CD) pipeline is an automated series of steps undertaken to get code from commit to production reliably.

The Solution: CI/CD Pipelines

- CI/CD pipelines can solve our challenges with `Set-Everything.ps1`

The Solution: CI/CD Pipelines

- CI/CD pipelines can solve our challenges with `Set-Everything.ps1`
  - **Ends Version Chaos**: Scripts are version controlled in Git – providing clear change history, approval gates, and defined versions. No more guessing which _v2 copy is the newest.

The Solution: CI/CD Pipelines

- CI/CD pipelines can solve our challenges with `Set-Everything.ps1`
  - **Ends Version Chaos**: Scripts are version controlled in Git – providing clear change history, approval gates, and defined versions. No more guessing which _v2 copy is the newest.
  - **Consistently Executes**: Runs scripts in a standardized agent environment, ensuring consistent PowerShell versions, modules, and permissions.

The Solution: CI/CD Pipelines

- CI/CD pipelines can solve our challenges with `Set-Everything.ps1`
  - **Ends Version Chaos**: Scripts are version controlled in Git – providing clear change history, approval gates, and defined versions. No more guessing which _v2 copy is the newest.
  - **Consistently Executes**: Runs scripts in a standardized agent environment, ensuring consistent PowerShell versions, modules, and permissions.
  - **No More Gambling with Prod**: Automated testing stages run before deployment, catching bugs safely before production.

The Solution: CI/CD Pipelines

- CI/CD pipelines can solve our challenges with `Set-Everything.ps1`
  - **Ends Version Chaos**: Scripts are version controlled in Git – providing clear change history, approval gates, and defined versions. No more guessing which _v2 copy is the newest.
  - **Consistently Executes**: Runs scripts in a standardized agent environment, ensuring consistent PowerShell versions, modules, and permissions.
  - **No More Gambling with Prod**: Automated testing stages run before deployment, catching bugs safely before production.
  - **Automates Triggers & Parameters:** Runs automatically upon code changes, targets servers systematically, and applies parameters reliably every time.

The Solution: CI/CD Pipelines

Choosing the Right Tool

- Other platforms exist to orchestrate PowerShell automations outside of pipelines:

Task Scheduler / cron jobs

OS built-in tools for scheduled script execution on a specific machine.

Best For:

- Simple, recurring tasks on individual servers *(e.g., nightly log cleanup).*
- Time-based triggers *(run at 2 AM daily).*

Choosing the Right Tool

- Other platforms exist to orchestrate PowerShell automations outside of pipelines:

Task Scheduler / cron jobs	Cloud Runbooks
	
OS built-in tools for scheduled script execution on a specific machine. **Best For**: • Simple, recurring tasks on individual servers *(e.g., nightly log cleanup).* • Time-based triggers *(run at 2 AM daily).*	Cloud platform services to centrally manage and run automation scripts. **Best For**: • Scheduled or event-driven tasks in the cloud *(VM start/stop, patching).* • Securely executing standardized scripts across cloud or hybrid fleets.

Choosing the Right Tool

- Other platforms exist to orchestrate PowerShell automations outside of pipelines:

Task Scheduler / cron jobs	Cloud Runbooks	Ad-hoc / Manual Execution
		
OS built-in tools for scheduled script execution on a specific machine. **Best For**: • Simple, recurring tasks on individual servers *(e.g., nightly log cleanup).* • Time-based triggers *(run at 2 AM daily).*	Cloud platform services to centrally manage and run automation scripts. **Best For**: • Scheduled or event-driven tasks in the cloud *(VM start/stop, patching).* • Securely executing standardized scripts across cloud or hybrid fleets.	Running scripts directly via console, Invoke-Command, RDP, etc. **Best For**: • One-off administrative tasks or quick fixes. • Interactive troubleshooting and script development.

Choosing the Right Tool

- Other platforms exist to orchestrate PowerShell automations outside of pipelines:

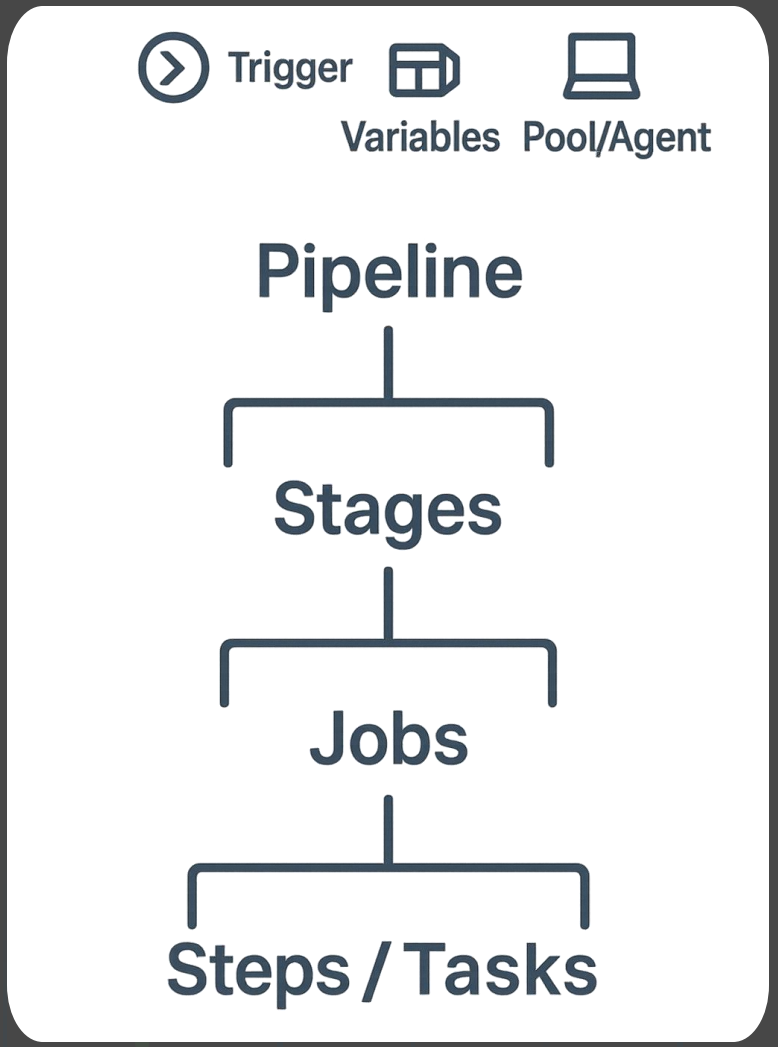
Task Scheduler / cron jobs	Cloud Runbooks	Ad-hoc / Manual Execution	Config Mgmt / Orchestrators
			
OS built-in tools for scheduled script execution on a specific machine. **Best For**: • Simple, recurring tasks on individual servers *(e.g., nightly log cleanup).* • Time-based triggers *(run at 2 AM daily).*	Cloud platform services to centrally manage and run automation scripts. **Best For**: • Scheduled or event-driven tasks in the cloud *(VM start/stop, patching).* • Securely executing standardized scripts across cloud or hybrid fleets.	Running scripts directly via console, Invoke-Command, RDP, etc. **Best For**: • One-off administrative tasks or quick fixes. • Interactive troubleshooting and script development.	Central platforms to define, schedule, and run automation jobs. (i.e. Ansible Tower) **Best For**: • Applying desired state configurations consistently. • Orchestrating scheduled operational workflows. • Providing self-service portals for tasks.

Choosing the Right Tool

- **Why Choose CI/CD Pipelines?**
  - Typically triggered by code changes and events but can be scheduled.
  - Orchestrate entire delivery workflows, not just single tasks.
  - Integrate deeply with source control, testing, secrets, artifacts, and cloud platforms.
  - Defined as code to ensure consistent, central, and audited execution.
- **Use Pipelines For:**
  - Tightly coupled development and deployment processes.
  - Work requiring high consistency, auditability, and visibility.
  - Complex packages that need build steps prior to deployment.

Anatomy of a Pipeline

- While some platforms offer UI based Pipeline development, most are defined as code (typically YAML) for version control.
- The common components of a pipeline are:
  - **Triggers**: Define when the pipeline runs.
  - **Variables**: Store reusable values like configuration settings or secrets used throughout the pipeline.
  - **Stages**: Major divisions of work (e.g., Build, Test, Deploy), allowing logical separation and dependencies.
  - **Jobs**: Units of execution within a Stage, each running on an agent. Jobs can often run in parallel.
  - **Steps / Tasks**: The individual commands or pre-packaged actions (like running PowerShell, .NET build, checking out code) executed sequentially within a job.



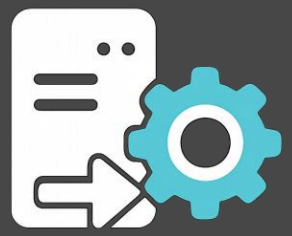
Where Does the Magic Happen? Pipeline Agents

- Pipeline Agents are the machines that execute your CI/CD jobs.
- Agents can be provider-managed or self-hosted:
  - **Cloud-hosted agents**
  - **Self-hosted agents**

- The execution context of your scripts is an important consideration when building your pipelines

Execution Context Matters!

- Where your PowerShell runs impacts network line-of-sight, dependency management, and potentially cost.

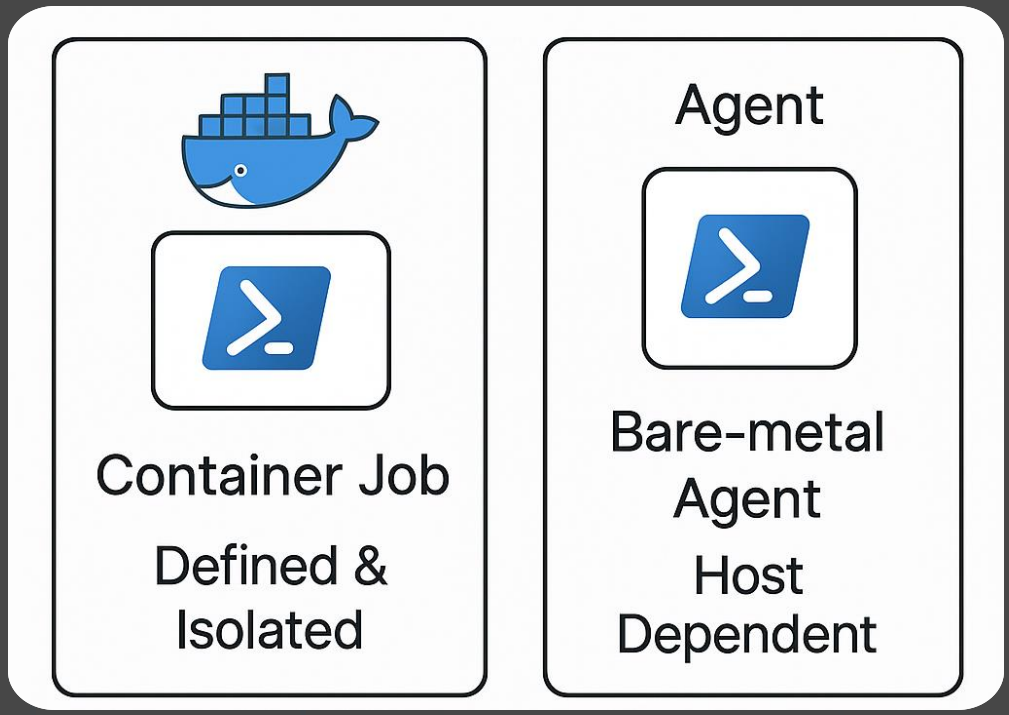


- You control the machine, OS, and installed tools. Offers direct access to your network resources but requires ongoing host & dependency management

- Ephemeral VMs with pre-defined environments (check provider docs for installed software!). Simpler setup, but limited customization and no inherent access to your private network

Run your Pipelines in Containers

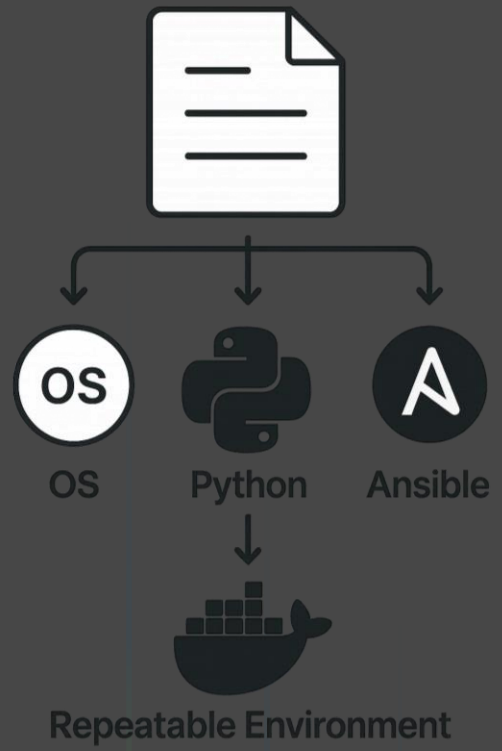
- Many CI/CD platforms let you specify that a job or step should run inside a Docker container.
- Why Choose Containers over Bare-metal?
  - **Consistency**: Guarantees the same runtime environment every single time, everywhere.
  - **Dependency Bundling**: Package all necessary tools and libraries within the container image itself.
  - **Isolation**: Avoid conflicts with software or versions installed on the host agent machine.



Using Dockerfiles

Using a Dockerfile, you can define the instructions to build your container: the base OS, required tools, libraries, and configurations

Example Scenario: Imagine needing Ansible and specific Python libraries for a configuration playbook. The Dockerfile installs exactly these components into the container image.



PowerShell 💙 Pipelines

- **It's Cross-Platform!**
  - With PowerShell 7+, your code can run across Windows, Linux, and macOS build agents.
- **The Automation "Glue"**
  - PowerShell includes native cmdlets for OS tasks, .NET integration, and excellent REST API interaction.
  - Other essential CLI tools can be orchestrated with PowerShell
- **Expansive Module Support**
  - Via modules, PowerShell offers countless capabilities for cloud (Azure Az, AWS Tools), Databases (DbaTools), Testing (Pester), Data Transformation, etc.
- **Leverage Familiar Skills Across Teams**
  - Your teams likely already have PowerShell skills! Empower your SysAdmins, Devs, and DevOps Engineers.

Pipeline Security Checklist

- **Verify Agent Identity:**
  - ☐ Understand the user account your agent runs as – its permissions dictate script access (local/network).
- **Secure Cloud Connections:**
  - ☐ **Never** hardcode credentials.
  - ☐ Use **Service Connections / OIDC / Managed Identities** for secure, auditable cloud API access.
- **Protect Secrets:**
  - ☐ Store sensitive data (API keys, passwords) in **Secrets Vaults**.
  - ☐ Leverage **secret masking** in logs (platform dependent).
- **Apply Least Privilege:**
  - ☐ Grant the agent account and service connections *only* the permissions essential for the task.

Pipeline Security Checklist

- **Verify Agent Identity:**
  - ☑ Understand the user account your agent runs as – its permissions dictate script access (local/network).
- **Secure Cloud Connections:**
  - ☑ **Never** hardcode credentials.
  - ☑ Use **Service Connections / OIDC / Managed Identities** for secure, auditable cloud API access.
- **Protect Secrets:**
  - ☑ Store sensitive data (API keys, passwords) in **Secrets Vaults**.
  - ☑ Leverage **secret masking** in logs (platform dependent).
- **Apply Least Privilege:**
  - ☑ Grant the agent account and service connections *only* the permissions essential for the task.

Platform Comparison

	Platform Overview	Key Reasons to Select Platform	PowerShell Experience
Azure DevOps	▪ Support managed and self-hosted agents ▪ Close integrations with Azure ▪ Flexible pipelines with integrated approvals	Existing .NET or Azure Footprint Great for Enterprise IT and Hybrid Cloud	Built-in PowerShell & Az PowerShell Tasks
	▪ Great integration with GitHub repositories ▪ Large community marketplace of Actions ▪ Managed runners or self-hosted	If code is hosted on GitHub Actions Marketplace	Actions & Pre-loaded PowerShell Core
	▪ Open-source and self-hosted ▪ Can orchestrate complex/legacy workflows ▪ Highly extensible	Open-Source & Self-hosted Highly extensible	PowerShell Usable if Installed on Agent
	▪ Supports hosted (GitLab.com) & self-managed instances ▪ Built-in Container Registry & Auto DevOps features accelerate pipeline setup	Unified DevOps Platform Strong Tool Integrations	Streamlined Integration with Docker, Official PowerShell Image

Platform Comparison

Each platform's pipeline syntax differs.

The following slides will showcase a simple pipeline example written for each platform:

- **Read a simple text file "version.txt" from the code repository**
- **Append the pipeline's Build ID to it**
- **Print the updated file content**

Azure DevOps

- **Built-in Tasks**: PowerShell@2 is the standard task
- **PowerShell Execution**: Can be specified in-line or via script file path
- **Variables**: Access pipeline variables like directly or map them as environment variables

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

pool:
  vmImage: 'windows-latest' # Or ubuntu-latest, pwsh works cross-platform

steps:
- task: PowerShell@2
  inputs:
    targetType: 'inline'
    script: |
      # Assume version.txt exists in the repo root with "Version: 1.0"
      $filePath = "$(System.DefaultWorkingDirectory)/version.txt"
      Write-Host "Reading from: $filePath"

      if (Test-Path $filePath) {
        $content = Get-Content $filePath
        $newContent = "$content - Build: $(Build.BuildId)"
        Write-Host "Updated Content: $newContent"
        # In a real scenario, you might write this back or use it
      } else {
        Write-Error "File not found: $filePath"
      }
    displayName: 'Read Version File and Append Build ID'
```

GitHub Actions

- **Checkout**: Uses an action to get your repository code onto the runner.
- **PowerShell Execution**: The **run** keyword executes commands.
- **Variables**: Access workflow information like run_id using expression syntax

```yaml
# .github/workflows/main.yml
name: Read Version Info

on: [push]

jobs:
  read_version:
    runs-on: ubuntu-latest # Or windows-latest, pwsh works cross-platform

    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Read Version File and Append Run ID
      shell: pwsh # Explicitly use PowerShell Core
      run: |
        # Assume version.txt exists in the repo root with "Version: 1.0"
        $filePath = "./version.txt" # Relative path in workspace
        Write-Host "Reading from: $filePath"

        if (Test-Path $filePath) {
          $content = Get-Content $filePath
          # Use GitHub context for the run ID
          $newContent = "$content - RunID: ${{ github.run_id }}"
          Write-Host "Updated Content: $newContent"
        } else {
          Write-Error "File not found: $filePath"
        }
```

Jenkins

- **Syntax**: Jenkins uses a Jenkinsfile (Groovy syntax)
- **PowerShell Step**: Use the PowerShell step to execute scripts on agents where PowerShell is available
- **Variables**: Access standard environment variables like $Env:BUILD_ID directly within PowerShell script

```
pipeline {
    agent { label 'windows' } // Assumes a Windows agent node tagged 'windows'

    stages {
        stage('Read Version Info') {
            steps {
                script {
                    // Use the 'powershell' step for Windows agents
                    powershell '''
                        # Assume version.txt exists in the workspace root with "Version: 1.0"
                        $filePath = "./version.txt"
                        Write-Host "Reading from: $filePath"

                        if (Test-Path $filePath) {
                            $content = Get-Content $filePath
                            # Use Jenkins built-in environment variable
                            $newContent = "$content - BuildID: $Env:BUILD_ID"
                            Write-Host "Updated Content: $newContent"
                        } else {
                            Write-Error "File not found: $filePath"
                        }
                    '''
                }
            }
        }
    }
}
```

Jenkins

- **Syntax**: Jenkins uses a Jenkinsfile (Groovy syntax)
- **PowerShell Step**: Use the PowerShell step to execute scripts on agents where PowerShell is available
- **Variables**: Access standard environment variables like $Env:BUILD_ID directly within PowerShell script

```
pipeline {
    agent { label 'windows'                                    l 'windows'

    stages {
        stage('Read Version
            steps {
                script {
                    // Use
                    powersh
                        # A                                    e root with "Version: 1.0"
                        $fi
                        Wri

                        if

                                                               able
                                                               BUILD_ID"
                                                               ."
                        } e

                        }
                    '''
                }
            }
        }
    }
}
```

GitLab CI

- **Script Execution**: Jobs define a script block containing shell commands.
- **Runner Config**: Use tags to select specific runners (e.g., a Windows runner where PowerShell is the default shell)
  - Or – use the image keyword to run the job in a container

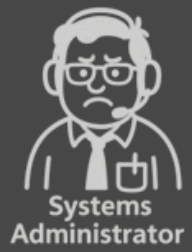
```yaml
# .gitlab-ci.yml
stages:
  - read_version

read_version_job:
  stage: read_version
  image: mcr.microsoft.com/powershell:latest # Use official PowerShell Docker image

  script:
    - echo "Running PowerShell script in GitLab CI..."
    # Assume version.txt exists in the repo root with "Version: 1.0"
    - $filePath = "./version.txt"
    - |
      if (Test-Path $filePath) {
        $content = Get-Content $filePath
        # Use GitLab CI predefined variable
        $newContent = "$content - PipelineIID: $env:CI_PIPELINE_IID"
        Write-Host "Updated Content: $newContent"
      } else {
        Write-Error "File not found: $filePath"
      }
```

Practical Examples

- PowerShell & CI/CD Pipelines can enhance workflows of IT professionals across a wide range of disciplines:

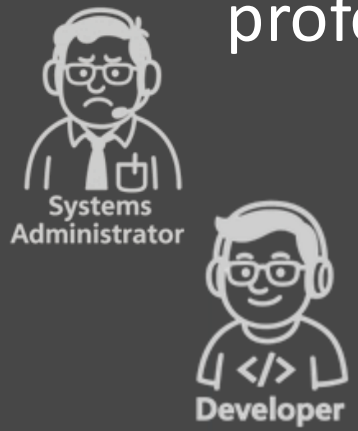
Practical Examples

- PowerShell & CI/CD Pipelines can enhance workflows of IT professionals across a wide range of disciplines:
  - **Systems Administrator**: Server configuration with PowerShell DSC



Practical Examples

- PowerShell & CI/CD Pipelines can enhance workflows of IT professionals across a wide range of disciplines:
  - **Systems Administrator**: Server configuration with PowerShell DSC
  - **Developer**: Python web application build and deployment



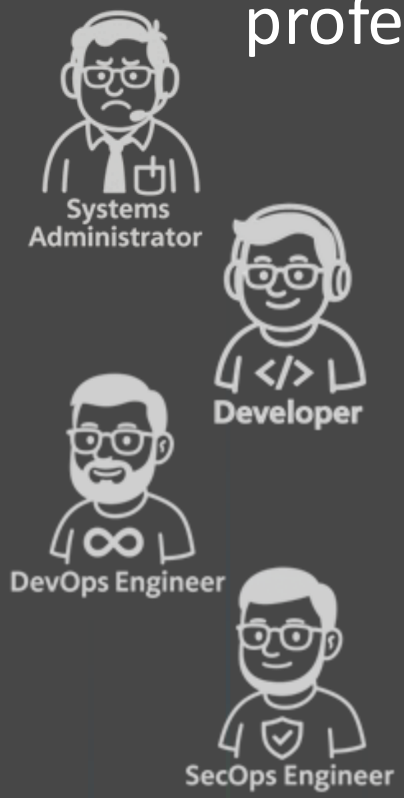
Practical Examples

- PowerShell & CI/CD Pipelines can enhance workflows of IT professionals across a wide range of disciplines:

  - **Systems Administrator**: Server configuration with PowerShell DSC
  - **Developer**: Python web application build and deployment
  - **DevOps Engineer**: Infrastructure as Code Deployment

Practical Examples

- PowerShell & CI/CD Pipelines can enhance workflows of IT professionals across a wide range of disciplines:



  - **Systems Administrator**: Server configuration with PowerShell DSC
  - **Developer**: Python web application build and deployment
  - **DevOps Engineer**: Infrastructure as Code Deployment
  - **SecOps Engineer**: Compliance Checks with Pester

Python Build and Deployment with Azure DevOps Pipelines

In this example, a developer automates the process of building, testing, and deploying a Python web application using an Azure DevOps YAML pipeline

Key Value:

- Build/deploy process defined as code, stored and versioned
- Ensures consistent, repeatable builds and deployments
- Automates quality gates through integrated testing steps

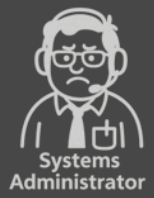
Infrastructure as Code Deployment with GitHub Actions

In this example, a DevOps Engineer uses GitHub Actions and PowerShell to deploy infrastructure to Azure

Key Value:

- Leverage Infrastructure as Code principals
- Enhance deployment consistency
- Automates infrastructure changes triggered by code commits

Server Configuration with DSC in a Jenkins Pipeline



In this example, a systems administrator uses Jenkins to apply a PowerShell Desired State Configuration (DSC) script to a target Windows server.

Key Value:

- Define Windows server configuration and version control with application code
- Accelerate application host configuration
- Extensibility allows for deep customization for legacy/on-prem

Security Compliance Checks in a GitLab Pipeline

In this example, a SecOps engineer automates security checks against Windows Server infrastructure using PowerShell within a GitLab CI/CD pipeline.

Key Value:

- Embeds security verification directly into the dev lifecycle
- Reduces the risk of deploying non-compliant configurations
- Automates enforcement of organizational security policies

Azure DevOps

DEMO

Link: https://github.com/Blakelishly/PipelineDemo

Conclusion

- ✓**Understand When & Why**: Learn the scenarios when PowerShell can be used in CI/CD pipelines to enhance your workflows.
- ✓**Explore Pipeline Deployment Methods**: Discover the various approaches for running PowerShell in pipelines—including the use of containers and custom Dockerfiles—and understand their associated considerations.
- ✓**Evaluate CI/CD Platforms**: Gain insights into the capabilities of different CI/CD tools for running PowerShell and how to select the best solution for your specific needs.

THANK YOU

Feedback is a gift. Please review this session