A Practical Master/Worker Framework for Windows Workgroup Networks Without Domain Services

Douglas C. Youvan

doug@youvan.com

August 14, 2025

Coordinating distributed computation on Windows machines in a workgroup environment—without the benefits of Active Directory or centralized policy management—presents a distinct set of challenges. While PsExec and similar remote execution tools promise headless control, they often fail in non-domain LANs due to local security policy restrictions, authentication issues, and inconsistent username conventions. In this work, we present a lightweight, reproducible master/worker framework that avoids these pitfalls by combining SMB file sharing, WinRM, and Python. The approach relies on a dedicated service account with uniform credentials across nodes, a shared folder for configuration and result exchange, and explicit path execution to avoid environment inconsistencies. This method eliminates the need for Group Policy edits or complex trust configurations, making it suitable for rapid prototyping and small-scale distributed workloads. We demonstrate the procedure using two AMD-based Windows systems, show the complete execution loop from master to worker and back, and outline how the setup can be extended to additional nodes with minimal effort. The solution is intentionally simple, transparent, and adaptable for future automation.

Keywords: Windows, workgroup, distributed computing, master-worker, SMB, WinRM, Python, PsExec alternative, remote execution, LAN. 44 pages. A collaboration with GPT-5.

Abstract

Running distributed computations across multiple Windows workstations in a workgroup environment—without Active Directory or centralized administration—presents unique challenges. Traditional remote execution tools such as PsExec are often hindered by local policy restrictions, authentication errors, and complex configuration steps. This work describes a lightweight, reproducible master/worker framework for CPU-intensive tasks that avoids PsExec entirely. The method uses SMB file sharing for configuration/result transfer, WinRM for optional remote management, and Python for computation. A dedicated service account with consistent credentials across machines enables predictable authentication, while explicit path execution ensures Python is invoked consistently regardless of PATH settings. We provide detailed step-by-step setup instructions, including PowerShell commands and Python code, and demonstrate the approach on two AMD-based Windows systems. The resulting framework is robust, easy to extend to new workers, and adaptable for later automation.

1. Introduction

Many Windows-based research and engineering environments operate in workgroup mode—a simple peer-to-peer network topology without domain controllers, centralized user accounts, or group policies. While workgroup setups are flexible and inexpensive, they complicate remote management, particularly for distributed computation.

Researchers and engineers often attempt to coordinate workstations using tools like PsExec, which in theory allow for headless, unattended execution on remote machines. However, PsExec's success depends heavily on consistent security policy settings, permissive network authentication modes, and matching credentials. In real-world workgroup networks, these conditions are rarely met:

- Local Security Policy Restrictions: Many Windows builds block "logon type" attempts from remote execution for non-domain users.

- Authentication Mismatches: Variations in username capitalization (e.g., Doug vs. doug) or machine-specific account names prevent proper credential matching.
- Administrative Complexity: Enabling PsExec often requires changes to Local Security Policy, registry keys, and firewall rules on every machine—changes which can break with Windows updates or security hardening.

In this paper, we propose a simpler, more reproducible master/worker system designed specifically for workgroup LANs. Instead of relying on direct remote command execution, our approach uses:

1. SMB File Sharing as the primary communication channel between master and worker.
2. A Dedicated Service Account with identical credentials on all machines to guarantee authentication success.
3. Python for portable computation, installed identically on each node via winget.
4. Explicit Path Execution to avoid “Python not found” issues caused by missing PATH entries or conflicting App Execution Aliases.

This method trades full headless automation for predictability and minimal setup time—workers can be added and producing results within minutes. Once the pipeline is proven reliable, the execution step can be automated with Windows Task Scheduler or lightweight monitoring scripts.

The remainder of this paper details the complete setup process, including all required PowerShell commands, Python code, SMB configuration, and installation notes, so that a reader can replicate this system from scratch on any small Windows workgroup.

---

2. Environment and Assumptions

2.1 Hardware and Network Layout

Our test environment consisted of two AMD Ser5 8C/16T small-form-factor PCs running Windows 10 Pro (though the method works with Windows 11 Pro as well).

- S5a (Master)
    - IP: 192.168.1.93
    - Role: Creates computation configuration, receives results from workers.
- S5b (Worker)
    - IP: 192.168.1.197
    - Role: Reads configuration from the master, performs CPU-intensive computation, writes results back.

Both are connected to the same LAN segment and can ping each other without packet loss.

2.2 Software Requirements

- Windows 10/11 Pro or equivalent edition with PowerShell 5.1+ (PowerShell 7.x optional but supported).
- Python 3.13.x installed via winget for consistency.
- Optional packages installed via pip (e.g., numpy or pandas) if the computation requires more than the Python standard library.

2.3 Account Model

We avoid using existing personal accounts (Doug, Administrator, etc.) to sidestep case mismatches and roaming profile issues. Instead, we create a dedicated service account with the same name and password on all nodes:

Username: cluster

Password: xxxxxxx

This account has local Administrator rights on each machine, ensuring it can read/write to the SMB share and execute tasks without privilege escalation issues.

---

3. Step-by-Step Setup

3.1 Creating the Service Account (Run on All Machines as Administrator)

net user cluster "Xxxxxxxx" /add

net localgroup "Administrators" cluster /add

If the account already exists, verify group membership with:

net localgroup "Administrators"

3.2 Setting Up the Master's Shared Folder (Run on S5a)

1. Create the shared directory:

New-Item -ItemType Directory -Path "C:\cluster" -Force

2. Share the folder with full access for cluster:

New-SmbShare -Name "cluster" -Path "C:\cluster" -FullAccess "cluster"

3. Allow SMB traffic through the firewall:

Enable-NetFirewallRule -DisplayGroup "File and Printer Sharing"

4. Set NTFS permissions:

icacls "C:\cluster" /grant "cluster:(OI)(CI)F" /T

3.3 Mapping the Master's Share from the Worker (Run on Worker)

From an elevated PowerShell window on S5b:

cmd /c 'net use Z: \\192.168.1.93\cluster /user:S5A\cluster "Xxxxxxxx" /persistent:yes'

Verify mapping:

Get-PSDrive Z

If successful, Z: will appear as a mapped drive pointing to \\192.168.1.93\cluster.

3.4 Installing and Verifying Python on All Machines

To ensure consistency across workers, we install the same Python version (3.13.x) via winget. This avoids path mismatches and guarantees an identical runtime environment.

Run the following in an elevated PowerShell window on each machine:

winget install -e --id Python.Python.3.13

Once installation completes, verify the binary location:

py -0p

Example output:

-V:3.13 * C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe

If py is missing, install the Python launcher:

winget install -e --id Python.Launcher

Check the version explicitly using the full path:

& "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" --version

This full-path invocation is critical in workgroup environments because relying on PATH variables or Windows' "App Execution Aliases" can lead to "Python not found" errors.

3.5 Installing Additional Python Packages

If the computation requires third-party packages (e.g., NumPy, Pandas, SciPy), install them explicitly using the same interpreter path:

& "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" -m pip install numpy pandas

Verify installation:

& "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" -m pip show numpy

This ensures all workers have identical dependencies.

4. Deploying the Worker Script

4.1 The worker.py File

Save the following Python script on each worker, ideally in the service account's home directory, e.g.,
C:\Users\cluster\worker.py.

```
import argparse, json, time, os, socket


MASTER_SHARE = r"Z:\\"  # Mapped SMB drive from master

CONFIG_PATH  = os.path.join(MASTER_SHARE, "run_config.json")


def cpu_intensive(seconds: int, seed: int) -> dict:
```

```
    total = 0
    x = seed or 1
    end = time.time() + seconds
    while time.time() < end:
        x = (1103515245 * x + 12345) & 0x7fffffff
        total ^= x
    return {"checksum": total}

def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--id", default=socket.gethostname())
    args = parser.parse_args()

    # Read config without BOM issues
    with open(CONFIG_PATH, "r", encoding="utf-8-sig") as f:
        cfg = json.load(f)

    minutes = int(cfg.get("minutes", 1))
    seed    = int(cfg.get("seed", 0))

    result = cpu_intensive(minutes * 60, seed)
    out = {
        "worker": args.id,
```

```
        "seed": seed,
        "minutes": minutes,
        "result": result,
        "status": "done",
        "host": socket.gethostname(),
    }

    out_path = os.path.join(MASTER_SHARE, f"results_{args.id}.json")
    with open(out_path, "w", encoding="utf-8") as f:
        json.dump(out, f)

if __name__ == "__main__":
    main()
```

4.2 Verifying the Worker Script

From the worker machine, confirm the file exists:

```
Get-ChildItem "C:\Users\cluster\worker.py"
```

5. Executing the Workflow

5.1 Create a Run Configuration on the Master

On S5a, create a configuration file in the shared folder:

```
[IO.File]::WriteAllText(
    "C:\cluster\run_config.json",
```

'{"seed":7000,"minutes":1}',

(New-Object System.Text.UTF8Encoding($false))

)

This avoids the UTF-8 BOM issue that can cause json.load to fail.

5.2 Run the Worker Script

On S5b (worker), execute:

& "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" "C:\Users\cluster\worker.py" --id S5b

The script will:

1. Read the configuration file from the master's share (Z:\run_config.json).
2. Run a CPU-intensive pseudo-random checksum loop for the specified number of minutes.
3. Write the result back to the master's share (Z:\results_S5b.json).

5.3 Verify Output on the Master

On S5a:

Get-Content C:\cluster\results_S5b.json

Example output:

{"worker": "S5b", "seed": 7000, "minutes": 1, "result": {"checksum": 1585469780}, "status": "done", "host": "S5b"}

6. Scaling to Multiple Workers

Once a single worker (S5b) is successfully integrated, adding more machines follows the same repeatable process.
The core principle is identical environment and credentials across all nodes, with minimal variation in configuration.

6.1 Naming and Identification

Each worker should have:

- A unique --id when running worker.py
- The same local account (cluster) with the same password (Xxxxxxxx)
- The same mapped SMB drive letter (Z:) pointing to the master's cluster share.

For example:

Worker Machine	Run Command
S5b	--id S5b
S5c	--id S5c
S5d	--id S5d

6.2 Adding a New Worker

1. Create the cluster Account (if not already present)
   Sign in as Administrator or existing admin, then in Admin PowerShell:
2. net user cluster "Xxxxxxxx" /add
3. net localgroup "Administrators" cluster /add
4. Install Python
5. winget install -e --id Python.Python.3.13

6. winget install -e --id Python.Launcher
7. Map the Master Share
8. cmd /c 'net use Z: \\192.168.1.93\cluster /user:S5A\cluster "Xxxxxxxx" /persistent:yes'
9. Copy the Worker Script
   Save worker.py to C:\Users\cluster\worker.py.
10. Verify Mapping and Script Presence
11. Test-Path Z:\run_config.json
12. Test-Path C:\Users\cluster\worker.py

6.3 Master Workflow with Multiple Workers

The master's run_config.json remains a single file in the shared folder, defining parameters for all workers.
Example:

{"seed":7000, "minutes":2}

All workers read the same file and independently write their own result files:

results_S5b.json

results_S5c.json

results_S5d.json

6.4 Parallel Launch

Currently, the system assumes manual starts on each worker.
On each worker:

& "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" "C:\Users\cluster\worker.py" --id S5c

(or change --id appropriately)

---

7. Automating Execution

Manual startup works for testing, but production-scale runs benefit from automation.

7.1 Scheduled Task Approach

A Scheduled Task can periodically check for the presence of a new run_config.json and launch the worker script automatically.

1. Create a small wrapper script (C:\Users\cluster\auto_worker.ps1):
2. $config = "Z:\run_config.json"
3. if (Test-Path $config) {
4. &
   "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" "C:\Users\cluster\worker.py" --id $env:COMPUTERNAME
5. }
6. Register Scheduled Task:
7. $action = New-ScheduledTaskAction -Execute "powershell.exe" -Argument "-File `"$env:USERPROFILE\auto_worker.ps1`""
8. $trigger = New-ScheduledTaskTrigger -Once -At (Get-Date).AddMinutes(1) -RepetitionInterval (New-TimeSpan -Minutes 5)
9. Register-ScheduledTask -Action $action -Trigger $trigger -TaskName "AutoWorker" -User "cluster" -Password "Xxxxxxxx"

This checks every 5 minutes for new work and executes automatically.

---

7. Automating Execution

Manual startup is fine for testing, but a scalable system benefits from automation. One easy method in Windows is to use the Task Scheduler so that each worker runs worker.py automatically whenever new work appears in the shared folder.

7.1 Scheduled Task Approach

Step 1 — Create a small PowerShell wrapper
Save this file as C:\Users\cluster\auto_worker.ps1 on each worker:

```
$config = "Z:\run_config.json"

if (Test-Path $config) {

    & "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" "C:\Users\cluster\worker.py" --id $env:COMPUTERNAME

}
```

This script checks if run_config.json exists on the mapped share (Z:) and, if so, runs the worker with the machine's hostname as its --id.

Step 2 — Create the Scheduled Task
In Admin PowerShell on each worker:

```
$action = New-ScheduledTaskAction -Execute "powershell.exe" -Argument "-File `"$env:USERPROFILE\auto_worker.ps1`""

$trigger = New-ScheduledTaskTrigger -Once -At (Get-Date).AddMinutes(1) -RepetitionInterval (New-TimeSpan -Minutes 5)

Register-ScheduledTask -Action $action -Trigger $trigger -TaskName "AutoWorker" -User "WORKER_ACCOUNT" -Password (Read-Host -Prompt "Enter password" -AsSecureString)
```

Replace WORKER_ACCOUNT with the local account name used on the worker (e.g., cluster).
When prompted, type the password securely — it won't be stored in your history.

Step 3 — Test the Automation

- Delete any existing result file for that worker from the master's share.
- Place a fresh run_config.json in the share.
- Wait for the scheduled interval; the worker should run automatically and produce a new result JSON.

8. Remote Triggering with WinRM (Master-initiated runs)

This section shows how to start workers remotely from the master (S5a) using PowerShell Remoting. It avoids the second-hop/credential delegation problem by explicitly mapping the master share on each worker inside the remote session with a share credential you provide at runtime.

8.1 One-time prerequisites

On each worker (once), in Admin PowerShell:

Enable-PSRemoting -Force

On S5a (master), in Admin PowerShell (trust specific workers by name/IP):

Use a comma-separated list of worker names/IPs you control.

Example for S5b and S5c:

winrm set winrm/config/client '@{TrustedHosts="S5b,S5c,192.168.1.197"}'

To confirm:

winrm get winrm/config/client

Note: In a workgroup, Kerberos isn’t available, so TrustedHosts is required for Negotiate/NTLM to work cleanly. Prefer listing just your workers, not “*”.

8.2 Master-side helper: write config (no BOM)

On S5a, write the run config once per batch (adjust values as needed):

```
[IO.File]::WriteAllText("C:\cluster\run_config.json",
  '{"seed":7000,"minutes":1}',
  (New-Object System.Text.UTF8Encoding($false)))
```

8.3 Master-side “remote trigger” script

Save this as C:\Users\cluster\master-trigger.ps1 on S5a (no secrets inside):

```
<#  master-trigger.ps1
    Remotely starts worker.py on one or more workers via WinRM.
    Prompts for two credentials at runtime:
      - Remote credential to log into the workers (e.g., local 'cluster' account)
      - Share credential used by workers to map \\<master>\cluster
#>

[CmdletBinding()]
param(
  [string[]]$Workers      = @("S5b"),         # computer names or IPs
  [string]  $MasterHost   = "S5a",            # hostname (not IP) for SMB /user:HOST\user
  [string]  $MasterIP     = "192.168.1.93",   # for \\IP\cluster UNC mapping
  [string]  $ShareName    = "cluster",        # shared folder name on master
```

```
  [string]  $PythonPath   =
"C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe",
  [string]  $WorkerScript = "C:\Users\cluster\worker.py",
  [int]     $TimeoutSec   = 1200            # per-worker wait for process completion
(optional)
)

Write-Host "Workers: $($Workers -join ', ')" -ForegroundColor Cyan
Write-Host "Master:  \\$MasterIP\$ShareName  (SMB)" -ForegroundColor Cyan

# Prompt for remote logon to workers (e.g., local 'cluster' on each worker)
$remoteCred = Get-Credential -Message "Enter the credential to log into workers
(e.g., WORKERNAME\cluster or just 'cluster')"

# Prompt for SMB share credential the workers will use to map the master share
$shareUser = Read-Host -Prompt "Enter master-share username (format:
$MasterHost\<user>)"
$sharePass = Read-Host -Prompt "Enter master-share password" -AsSecureString
$shareCred = New-Object
System.Management.Automation.PSCredential($shareUser, $sharePass)

$script = {
  param($MasterIP, $ShareName, $MasterHost, $PythonPath, $WorkerScript,
$ShareCred, $TimeoutSec)
```

```
$ErrorActionPreference = 'Stop'
$root = "\\$MasterIP\$ShareName"

# Ensure Z: is free then map it with explicit credentials
try { cmd /c 'net use Z: /delete /y' | Out-Null } catch {}
$cmd = "net use Z: $root /user:$($ShareCred.UserName) <password> /persistent:yes"

# Build a one-off CMD line with a temporary plain text password value derived securely at runtime.
# (We avoid sending secure string over the wire; only the remote host sees the password to authenticate to the master.)
$bstr = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($ShareCred.Password)
$plain = [Runtime.InteropServices.Marshal]::PtrToStringBSTR($bstr)
try {
  cmd /c $cmd.Replace("<password>", '"' + $plain + '"') | Out-Null
} finally {
  if ($bstr -ne [IntPtr]::Zero) { [Runtime.InteropServices.Marshal]::ZeroFreeBSTR($bstr) }
}

# Confirm mapping
$z = Get-PSDrive -Name Z -ErrorAction SilentlyContinue
if (-not $z) { throw "Failed to map Z: to $root" }
```

```
# Ensure worker script exists
if (-not (Test-Path $WorkerScript)) {
  throw "Worker script not found at $WorkerScript"
}
if (-not (Test-Path "Z:\run_config.json")) {
  throw "No run_config.json found on Z:\"
}

# Run the worker and wait
$psi = New-Object System.Diagnostics.ProcessStartInfo
$psi.FileName  = $PythonPath
$psi.Arguments = "`"$WorkerScript`" --id $env:COMPUTERNAME"
$psi.UseShellExecute = $false
$psi.RedirectStandardOutput = $true
$psi.RedirectStandardError  = $true

$p = [System.Diagnostics.Process]::Start($psi)
if (-not $p) { throw "Failed to start worker process" }

if ($TimeoutSec -gt 0) {
  if (-not $p.WaitForExit($TimeoutSec * 1000)) {
    try { $p.Kill() } catch {}
```

```
    throw "Worker timed out after $TimeoutSec seconds"
   }
  } else {
   $p.WaitForExit()
  }

  # Return a small status object
  [PSCustomObject]@{
   ComputerName = $env:COMPUTERNAME
   ExitCode     = $p.ExitCode
  }
}

# Invoke on all workers
Invoke-Command -ComputerName $Workers `
  -ScriptBlock $script `
  -Credential $remoteCred `
  -Authentication Negotiate `
  -ArgumentList $MasterIP, $ShareName, $MasterHost, $PythonPath,
$WorkerScript, $shareCred, $TimeoutSec
```

What this does

- Prompts once for the remote credential (to log into each worker via WinRM).

- Prompts once for the SMB share credential (used on the workers to map \\<MasterIP>\cluster inside the remote session).
- On each worker, it:
    - Maps Z: to the master share using the provided share credential.
    - Verifies worker.py and run_config.json exist.
    - Runs worker.py via the explicit Python path with --id $env:COMPUTERNAME.
    - Returns a small status object.

8.4 Example usage

1. Ensure the master has written a config:

```
[IO.File]::WriteAllText("C:\cluster\run_config.json",
  '{"seed":1234,"minutes":2}',
  (New-Object System.Text.UTF8Encoding($false)))
```

2. Trigger two workers from S5a:

```
powershell -File C:\Users\cluster\master-trigger.ps1 -Workers S5b,S5c -
MasterHost S5a -MasterIP 192.168.1.93
```

You'll be prompted twice:

- Once for the remote login to workers (use the workers' local account, e.g., cluster with its password).
- Once for the share login seen by workers when mapping Z: (use an account allowed on the master share, e.g., S5a\cluster).

3. Verify results on S5a:

```
Get-Content C:\cluster\results_S5b.json
Get-Content C:\cluster\results_S5c.json
```

8.5 Notes and options

- If you prefer to avoid converting a secure string to plain text on the worker for net use, replace the cmd /c net use mapping with:
- New-PSDrive -Name Z -PSProvider FileSystem -Root "\\$MasterIP\$ShareName" -Persist -Credential $ShareCred

In some environments, New-PSDrive -Persist under remoting can be finicky; net use is typically more reliable.

- If Invoke-Command returns “Access is denied” or trust issues:
    - Recheck TrustedHosts on S5a to include each worker.
    - Confirm Enable-PSRemoting -Force has been run on the worker.
    - Use -Authentication Negotiate (as above).
- If you need to pass a specific --id, change --id $env:COMPUTERNAME in the script to use any string you like, or pass it in via -ArgumentList.

9. Security Considerations

This approach uses workgroup-mode PowerShell Remoting (NTLM over HTTP, TCP 5985) and SMB shares to move data between the master and workers. While this is ideal for a trusted LAN testbed, you should be aware of the risks and ways to limit exposure.

9.1 TrustedHosts and authentication

- In non-domain networks, Kerberos authentication is not available, so NTLM with TrustedHosts is used.
- Setting TrustedHosts = * would allow WinRM connections to/from any host. For security, explicitly list the worker names or IP addresses you control:
- winrm set winrm/config/client '@{TrustedHosts="S5b,192.168.1.197"}'

- Remember that TrustedHosts only relaxes authentication checks — it does not encrypt the traffic by itself.

9.2 Firewall scope

- When enabling File and Printer Sharing or Remote Desktop, restrict the firewall rule scope to your LAN subnet.
- Example to scope SMB to the 192.168.1.0/24 network:
- Set-NetFirewallRule -DisplayGroup "File and Printer Sharing" -RemoteAddress 192.168.1.0/24

9.3 SMB credentials

- Workers must authenticate to the master's SMB share.
- Create a dedicated share account (e.g., cluster) with minimal rights — only Change/Write on the C:\cluster folder, no broader administrative rights.
- This prevents a worker compromise from giving unnecessary access to other files.

9.4 WinRM encryption

- By default, NTLM over HTTP does encrypt traffic using message-level encryption, but it does not prevent certain relay attacks.
- For higher security, consider using WinRM over HTTPS (TCP 5986) with self-signed or internal CA certificates.

9.5 Least privilege

- The cluster account used for running worker.py does not need local administrator rights unless installing packages or adjusting system configuration.
- On the master, only give Full Control on the C:\cluster directory to accounts that must write results.

---

10. Discussion and Future Work

The method we have documented demonstrates that a fully functional distributed job execution environment can be set up in a Windows workgroup setting without additional orchestration frameworks or expensive infrastructure. By leveraging built-in Windows capabilities — PowerShell Remoting, SMB shares, and basic firewall configuration — along with Python for portable computation, we achieve an arrangement that can be extended across multiple machines with minimal setup time once the procedure is well understood.

10.1 Observations from implementation

- Reliability: Once WinRM and SMB are correctly configured, job dispatch and result retrieval were consistent, even across reboots, as long as share mappings and firewall rules persisted.
- Bottlenecks: SMB shares on a single master can become I/O bound if many workers write results simultaneously, particularly with larger outputs.
- Security trade-offs: TrustedHosts and NTLM authentication are acceptable in an isolated LAN but are inappropriate for untrusted or routed networks unless combined with HTTPS and stronger credential handling.
- User account consistency: Using a dedicated, identically named cluster account on each machine simplified credential management and avoided cross-user confusion (e.g., doug vs. doug.S5B).

10.2 Areas for improvement

1. Parallel job dispatch
   Instead of manually triggering workers, the master could use a simple queue file or JSON list in C:\cluster that workers poll at intervals.
2. Result aggregation
   The master's script could automatically combine all results_<ID>.json files into a single CSV or database row for quick analysis.

3. Python environment management
   To avoid per-machine package drift, consider using a pre-configured virtual environment or distributing the Python executable with all dependencies.
4. Transport security
   Transition to WinRM over HTTPS with self-signed certificates and restricted firewall scopes to protect credentials and data in transit.
5. Cross-platform considerations
   Although this work targets Windows, the architecture could be adapted to mixed OS networks using SSH for Linux/Mac workers and SMB/NFS for shared storage.

11. Conclusion

This work presents a practical, reproducible method for building a small-scale distributed computation system entirely with tools already available in modern Windows installations.
By combining PowerShell Remoting for command execution, SMB shares for lightweight file-based communication, and Python for portable computation logic, we demonstrate that researchers and engineers can achieve meaningful parallelism across commodity hardware in a workgroup setting — without relying on domain controllers, commercial orchestration software, or complex cloud infrastructure.

The key to our success was simplifying the approach after initial overcomplication:

- We avoided PsExec policy pitfalls by using WinRM and explicit credential mapping.
- We standardized user accounts (cluster) across all nodes to remove credential ambiguity.
- We validated each step — network reachability, share access, Python installation — before moving to the next.

Our setup can be reproduced quickly once the process is documented, making it a valuable baseline for labs, small research groups, or teaching environments where a modest number of Windows PCs can be harnessed for parallel workloads.

Future work will focus on security hardening, automated job dispatch, and cross-platform integration to broaden the applicability of this method. By publishing these instructions openly, we aim to save others the troubleshooting effort we faced, and enable them to start with a proven, functioning configuration.

References

[1] United States v. Microsoft Corp., 253 F.3d 34 (D.C. Cir. 2001). *The seminal case in which the court concluded that the noble art of "giving users exactly one choice" might be a problem.*

[2] European Commission, "Commission Fines Microsoft for Tying Web Browser to Operating System," Brussels, 2009. *The EU gently explained that "choice" is not defined as "click here to install Edge instead of Edge."*

[3] United States Department of Justice, "Proposed Final Judgment in Microsoft Antitrust Case," 2002. *Otherwise known as: "Don't do that again… unless it's profitable."*

[4] The People of the United States v. Microsoft, circa 1998. *Historic moment when tech journalists got to explain the phrase "monopoly" to the public by pointing at the Windows 98 desktop.*

[5] "Netscape Navigator: Rise and Fall," Journal of Forgotten Software, vol. 1, no. 1, 2005. *Primary victim of the 'free with Windows' business model.*

[6] European Union v. Microsoft (Windows Media Player Case), 2004. *Because sometimes you need to be fined twice to get the message.*

[7] Hypothetical Future Case, *Researchers v. Microsoft*, 2025. *Alleging emotional distress from unannounced UI changes during live demos.*

Appendix A: Troubleshooting

This appendix captures common issues encountered during development and testing of the master/worker setup in a Windows workgroup environment, along with fixes.

A.1 SMB share not accessible from worker

Symptoms

- net view \\MASTER returns *System error 5 has occurred. Access is denied.*
- ii \\MASTER\cluster fails with *path does not exist*.

Fixes

1. Verify the share exists on the master:
2. Get-SmbShare -Name cluster
3. Ensure the dedicated share account exists and has rights:
4. Grant-SmbShareAccess -Name cluster -AccountName "MASTER\cluster" -AccessRight Change -Force
5. icacls "C:\cluster" /grant "MASTER\cluster:(OI)(CI)M" /T
6. Map the drive from the worker using explicit credentials:
7. cmd /c 'net use Z: \\192.168.1.93\cluster /user:MASTER\cluster "YOUR_PASSWORD" /persistent:yes'

A.2 WinRM remoting fails with TrustedHosts error

Symptoms

- Enter-PSSession gives *The WinRM client cannot process the request....*

Fixes

1. On both master and worker, run in Admin PowerShell:
2. winrm quickconfig -q
3. winrm set winrm/config/client '@{TrustedHosts="192.168.1.197"}'

Replace 192.168.1.197 with the actual worker IP.

4. Re-test:
5. Test-WsMan 192.168.1.197

A.3 Python not found on worker

Symptoms

- python --version or py -3.13 --version fails.

Fixes

1. Install Python via Winget:
2. winget install -e --id Python.Python.3.13
3. winget install -e --id Python.Launcher
4. Confirm path:
5. py -0p
6. Use explicit path in scripts:
7. & "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" script.py

---

A.4 UTF-8 BOM issue in run_config.json

Symptoms

- Worker crashes with:

- json.decoder.JSONDecodeError: Unexpected UTF-8 BOM

Fix
Write the config without BOM:

[IO.File]::WriteAllText("C:\cluster\run_config.json", '{"seed":7000,"minutes":1}', (New-Object System.Text.UTF8Encoding($false)))

A.5 File open/write errors

Symptoms

- io.UnsupportedOperation: not writable when worker writes results.

Fixes

- Ensure the result file is opened with write mode ("w") in Python.
- Confirm the worker account has Change or Full access to the SMB share and underlying NTFS path.
- Appendix B — Complete PowerShell Setup Scripts

These scripts assume:

- Machines are in the same workgroup and LAN.
- You have a dedicated account named cluster on all machines, same password.
- You will substitute your own password where indicated.

Appendix B: Master Node (S5a) Setup Script

Run in PowerShell as Administrator:

```
# === Create working folder ===
New-Item -ItemType Directory -Path "C:\cluster" -Force | Out-Null

# === Create or reset dedicated share account ===
net user cluster "YOUR_PASSWORD" /add
net localgroup "Users" cluster /add

# === Share the folder over SMB ===
if (-not (Get-SmbShare -Name cluster -ErrorAction SilentlyContinue)) {
    New-SmbShare -Name "cluster" -Path "C:\cluster" -FullAccess "S5a\cluster"
}

# === Permissions ===
Grant-SmbShareAccess -Name cluster -AccountName "S5a\cluster" -AccessRight
Change -Force
icacls "C:\cluster" /grant "S5a\cluster:(OI)(CI)M" /T

# === Open firewall for SMB ===
Enable-NetFirewallRule -DisplayGroup "File and Printer Sharing"

# === Enable WinRM ===
```

Enable-PSRemoting -Force

winrm set winrm/config/client '@{TrustedHosts="192.168.1.197"}' # Add worker IP(s)

Write-Host "Master setup complete."

B.2 Worker Node (S5b) Setup Script

Run in PowerShell as Administrator:

=== Create or reset dedicated share account ===

net user cluster "YOUR_PASSWORD" /add

net localgroup "Administrators" cluster /add

=== Enable WinRM ===

Enable-PSRemoting -Force

winrm set winrm/config/client '@{TrustedHosts="192.168.1.93"}' # Add master IP

=== Map the master share ===

cmd /c 'net use Z: \\192.168.1.93\cluster /user:S5A\cluster "YOUR_PASSWORD" /persistent:yes'

=== Install Python 3.13 and Launcher ===

winget install -e --id Python.Python.3.13

winget install -e --id Python.Launcher

=== Verify Python ===

py -0p

Write-Host "Worker setup complete. Save worker.py in C:\Users\cluster\."

B.3 First Run Test

On master:

[IO.File]::WriteAllText("C:\cluster\run_config.json", '{"seed":7000,"minutes":1}', (New-Object System.Text.UTF8Encoding($false)))

On worker:

& "C:\Users\cluster\AppData\Local\Programs\Python\Python313\python.exe" "C:\Users\cluster\worker.py" --id S5b

On master, verify results:

Get-Content C:\cluster\results_S5b.json

Appendix C: Worker and Master Scripts (Complete, Ready to Use)

This appendix includes the complete, minimal code artifacts you can drop into place on the worker(s) and master. The worker script is self-contained and robust to UTF-8 BOM in the config. The master helper writes a clean config (UTF-8 without BOM) and waits for results.

C.1 worker.py (place on each worker)

Path on worker:
C:\Users\cluster\worker.py

import argparse, json, time, os, socket

```
MASTER_SHARE = r"Z:\\"
CONFIG_PATH  = os.path.join(MASTER_SHARE, "run_config.json")


def cpu_intensive(seconds: int, seed: int) -> dict:
    total = 0
    x = seed or 1
    end = time.time() + seconds
    # Tight pseudo-random bit-mixing loop; CPU-bound; deterministic for given seed+duration
    while time.time() < end:
        x = (1103515245 * x + 12345) & 0x7fffffff
        total ^= x
    return {"checksum": total}


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--id", default=socket.gethostname())
    args = parser.parse_args()

    # Tolerate BOM or plain UTF-8 in the config
    with open(CONFIG_PATH, "r", encoding="utf-8-sig") as f:
        cfg = json.load(f)
```

```
    minutes = int(cfg.get("minutes", 1))
    seed    = int(cfg.get("seed", 0))

    result = cpu_intensive(minutes * 60, seed)
    out = {
        "worker": args.id,
        "seed": seed,
        "minutes": minutes,
        "result": result,
        "status": "done",
        "host": socket.gethostname(),
    }

    out_path = os.path.join(MASTER_SHARE, f"results_{args.id}.json")
    # Write UTF-8 JSON
    with open(out_path, "w", encoding="utf-8") as f:
        json.dump(out, f)

if __name__ == "__main__":
    main()
```

Notes:

- Expects the master share to be mapped as drive Z: pointing to \\<MASTER>\cluster.

- The worker reads Z:\run_config.json and writes Z:\results_<ID>.json.
- Use --id <Name> to control the result filename.

C.2 Master helper: write config and wait for results

Path on master (S5a):
%USERPROFILE%\master-run.ps1 (example: C:\Users\cluster\master-run.ps1)

```
<#  master-run.ps1

   Writes C:\cluster\run_config.json (UTF-8 without BOM) and waits for
results_<WorkerId>.json.

#>

[CmdletBinding()]

param(

  [int]$Minutes = 1,

  [int]$Seed = 7000,

  [string[]]$WorkerId = @("S5b"),

  [int]$TimeoutSec = 600,

  [switch]$ClearOldResults

)


$ErrorActionPreference = "Stop"

$ClusterPath = "C:\cluster"

$ConfigPath  = Join-Path $ClusterPath "run_config.json"


New-Item -ItemType Directory -Path $ClusterPath -Force | Out-Null
```

```
if ($ClearOldResults) {
  foreach ($id in $WorkerId) {
    $p = Join-Path $ClusterPath "results_$id.json"
    if (Test-Path $p) { Remove-Item $p -Force }
  }
}

# Write config as UTF-8 without BOM
$utf8NoBom = New-Object System.Text.UTF8Encoding($false)
$configObj = @{ seed = $Seed; minutes = $Minutes }
[IO.File]::WriteAllText($ConfigPath, ($configObj | ConvertTo-Json -Compress), $utf8NoBom)
Write-Host "Wrote $ConfigPath : $(Get-Content $ConfigPath)" -ForegroundColor Cyan

function Wait-Result {
  param([string]$Id, [int]$Timeout = 600)
  $resultPath = Join-Path $ClusterPath "results_$Id.json"
  $sw = [Diagnostics.Stopwatch]::StartNew()
  while ($sw.Elapsed.TotalSeconds -lt $Timeout) {
    if (Test-Path $resultPath) {
      Write-Host "[$Id] Result file found -> $resultPath" -ForegroundColor Green
      Get-Content $resultPath
```

```
      return $true
    }
    Start-Sleep -Seconds 1
  }
  Write-Warning "[$Id] Timed out waiting for $resultPath"
  return $false
}

$allOk = $true
foreach ($id in $WorkerId) {
  $ok = Wait-Result -Id $id -Timeout $TimeoutSec
  if (-not $ok) { $allOk = $false }
}
if ($allOk) {
  Write-Host "All expected results arrived." -ForegroundColor Green
} else {
  Write-Warning "One or more results did not arrive in time."
}
```

Usage:

```
# One worker, 1 minute
powershell -File $HOME\master-run.ps1 -Minutes 1 -Seed 7000 -WorkerId S5b -TimeoutSec 300 -ClearOldResults
```

Multiple workers

powershell -File $HOME\master-run.ps1 -Minutes 5 -Seed 42 -WorkerId S5b,S5c -TimeoutSec 1200 -ClearOldResults

---

C.3 Optional: worker bootstrap script (single-run setup on a new worker)

If you want to automate Python installation, share mapping, and a smoke test on a fresh worker, use the following. It does not embed credentials; you pass them at run time or edit before use.

Path on worker:
%USERPROFILE%\setup-worker.ps1

```
param(
  [string]$Master       = "192.168.1.93",   # Master IP or hostname for \\Master\cluster
  [string]$MasterLogin   = "S5A",           # Hostname used in /user:HOST\user when mapping
  [string]$ShareName     = "cluster",
  [string]$WorkerId      = $env:COMPUTERNAME
)

$ErrorActionPreference = "Stop"

# Admin check
$principal = New-Object Security.Principal.WindowsPrincipal([Security.Principal.WindowsIdentity]::GetCurrent())
```

```
if (-not
$principal.IsInRole([Security.Principal.WindowsBuiltinRole]::Administrator)) {
  throw "Run this script in PowerShell as Administrator."
}

# Prompt for share credential securely (e.g., S5A\cluster)
$shareUser = Read-Host -Prompt "Enter share username (e.g.,
$MasterLogin\cluster)"
$sharePass = Read-Host -Prompt "Enter share password" -AsSecureString
$shareCred = New-Object
System.Management.Automation.PSCredential($shareUser, $sharePass)

# Ensure Python 3.13 and Launcher
try { $null = & py -3.13 --version 2>$null } catch {
  & winget install -e --id Python.Python.3.13 --accept-package-agreements --
accept-source-agreements | Out-Null
  & winget install -e --id Python.Launcher      --accept-package-agreements --
accept-source-agreements | Out-Null
}

# Find python.exe
function Get-Python313Path {
  try {
    $paths = & py -0p 2>$null
    if ($paths) { foreach ($p in $paths) { if ($p -match "3\.13") { return $p.Trim() } } }
```

```
  } catch {}
  $candidates = @(
    "$env:LOCALAPPDATA\Programs\Python\Python313\python.exe",
    "C:\Program Files\Python313\python.exe"
  )
  foreach ($c in $candidates) { if (Test-Path $c) { return $c } }
  return $null
}
$PythonExe = Get-Python313Path
if (-not $PythonExe) { throw "Could not locate Python 3.13; reopen a new Admin PowerShell and rerun." }

# Map master share as Z:
try { cmd /c 'net use Z: /delete /y' | Out-Null } catch {}
$root = "\\$Master\$ShareName"
$bstr = [Runtime.InteropServices.Marshal]::SecureStringToBSTR($shareCred.Password)
$plain = [Runtime.InteropServices.Marshal]::PtrToStringBSTR($bstr)
try {
  $cmd = "net use Z: $root /user:$($shareCred.UserName) `"$plain`" /persistent:yes"
  cmd /c $cmd | Out-Null
} finally {
```

```
  if ($bstr -ne [IntPtr]::Zero) {
[Runtime.InteropServices.Marshal]::ZeroFreeBSTR($bstr) }
}
if (-not (Get-PSDrive -Name Z -ErrorAction SilentlyContinue)) { throw "Failed to map Z: to $root" }


# Write worker.py
$workerPath = Join-Path $env:USERPROFILE "worker.py"
$workerCode = @'
import argparse, json, time, os, socket
MASTER_SHARE = r"Z:\\"
CONFIG_PATH  = os.path.join(MASTER_SHARE, "run_config.json")
def cpu_intensive(seconds: int, seed: int) -> dict:
    total = 0; x = seed or 1; end = time.time() + seconds
    while time.time() < end:
        x = (1103515245 * x + 12345) & 0x7fffffff
        total ^= x
    return {"checksum": total}
def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--id", default=socket.gethostname())
    args = parser.parse_args()
    with open(CONFIG_PATH, "r", encoding="utf-8-sig") as f:
        cfg = json.load(f)
```

```
    minutes = int(cfg.get("minutes", 1)); seed = int(cfg.get("seed", 0))
    result = cpu_intensive(minutes * 60, seed)
    out = {"worker": args.id, "seed": seed, "minutes": minutes, "result": result,
"status": "done", "host": socket.gethostname()}
    out_path = os.path.join(MASTER_SHARE, f"results_{args.id}.json")
    with open(out_path, "w", encoding="utf-8") as f: json.dump(out, f)
if __name__ == "__main__": main()
'@
[IO.File]::WriteAllText($workerPath, $workerCode, (New-Object
System.Text.UTF8Encoding($false)))
Write-Host "worker.py -> $workerPath"

# Smoke test
& "$PythonExe" "$workerPath" --id $WorkerId
Write-Host "Done. Expect: \\$Master\$ShareName\results_${WorkerId}.json"
```

Usage:

```
# Admin PowerShell on the worker:
powershell -ExecutionPolicy Bypass -File "$env:USERPROFILE\setup-worker.ps1" -
Master 192.168.1.93 -MasterLogin S5A -WorkerId S5b
```

You will be prompted for the share credential at runtime (no password in the script or command line).

C.4 Optional: master remote-trigger script (credential-safe)

If you want the master to start workers via WinRM on demand, use the script from the main text (Section 8, remote triggering), which prompts for credentials at runtime and avoids embedding secrets.