KNOWLEDGE GAP

DETONATOR

THE TOOL

CLAUDE CODE

MANAGING

THE TARGET

AZURE LOCAL

V1 Configuration and Cluster Management
Using SSH over Azure Arc

The Rules. The Hooks. The Gotchas. The Lockout.

JOURNAL

Dustin Winkler
AWACS LLC
awacs.ai

KNOWLEDGE GAP: DETONATOR

Claude Code Managing Azure Local

V1 Configuration and Cluster Management Using SSH over Azure Arc

The Rules, The Hooks, The Gotchas, and The Lockout

Book 1 in the Knowledge Gap: Detonator series

Dustin Winkler

Founder, AWACS LLC

awacs.ai

> *This is not a success story. Or rather, it is — but only because it was also a failure story first. The failures are the interesting part. That's why I wrote it down.*

What This Is

This is a field journal. Not a manual. Not a best practices guide. Not a whitepaper.

In March 2026, I deployed enterprise third-party software on four WDAC-enforced Azure Local management nodes in a remote data center from the other side of the world. I used Claude Code as my operations partner for the entire engagement. I had no remote access, no documentation, and no validated credentials when I started.

Along the way, the AI locked me out of production. Twice. In the same session. I recovered laterally through a sibling node using infrastructure knowledge the AI didn't have. Then I wrote rules so it couldn't happen the same way again.

This book documents exactly what happened: the decisions, the failures, the recovery, and the governance framework that emerged from the wreckage. Every rule traces to a specific incident with a specific date.

If you're running Azure Local and wondering whether AI can actually help with infrastructure operations, this is your answer. Yes. But only if you know where the trust boundary is. This book shows you where I found it.

Who this is for

- Infrastructure engineers running Azure Local or Azure Stack HCI
- Anyone using or considering Claude Code for production operations
- Engineers who want to see what AI-assisted infrastructure work actually looks like, not what vendors say it looks like

Who this is not for

- People looking for a Terraform tutorial or Azure certification prep
- People who want theory without incidents
- People who think AI in infrastructure is either magic or impossible

Table of Contents

CHAPTER 1

The Problem

Zero Access, Zero Documentation, A Cluster in a Remote Data Center

I had four nodes. Azure Local management nodes in a remote data center. The mission was straightforward on paper: deploy enterprise third-party software on all four nodes so we could protect the HyperV VMs running on the cluster.

The nodes were running Windows Defender Application Control in Enforced mode — the default security posture for Azure Local. WDAC blocks anything that isn't explicitly on the allowlist. The software we needed to install wasn't on the allowlist. So even if I could get onto the nodes, I'd need to deal with WDAC before the agent would run.

That was the mission. It sounded like an afternoon's work.

It was not an afternoon's work.

The Starting State

Let me be honest about what "zero" actually looked like.

Zero access. WinRM blocked, no VPN, no RDP, no out-of-band management. Every conventional remote access path was closed.

Zero validated credentials. I had a service account name and password stored in a spreadsheet. I had no idea if they were current. I had no way to test them without access.

Zero documentation. The vendor's documentation for Azure Local required logging into their support portal. The vendor KB was effectively inaccessible.

This is not an unusual starting state for infrastructure work. It's actually pretty common. The difference was that I was going to use Claude Code as my operations partner for the entire engagement — not just for code generation, but as an active participant in the problem-solving process.

The First Session: Building a Map Without a Compass

Session 1 was research. No live commands. No production contact. Just: what do we know, and how do we fill the gaps?

Claude compiled a 530-line knowledge base from training data: Azure Local architecture, WDAC policy deployment procedures, vendor compatibility documentation, known failure patterns. None of it was authoritative. All of it was good enough to build a plan on, as long as we validated before acting.

One of the first useful outputs was a compilation of what we didn't know:

- Whether the service account credentials were current
- Whether Azure Arc SSH was enabled on any of the nodes
- Whether the WDAC cmdlets we'd need were available (they're module-specific and not always present)

- Whether the management appliance was reachable from the nodes over the local network
- Whether the WDAC policy XML we had (obtained through the vendor's support channel) would actually work

I had obtained a vendor-provided document describing a FilePath-based supplemental WDAC policy for exactly this situation. It hadn't been published publicly. We put it into Claude's context. That document became the key — without it, there was no path forward on WDAC.

The session ended with a deployment plan: twelve steps, each with an expected output and a failure recovery path. None of it had been tested. But we had something to execute against.

The Failed Access Attempts

Session 2 opened with the first live commands. The plan was to establish access to node-01, validate credentials, and begin the WDAC deployment.

Three access methods failed before we found the one that worked.

Attempt 1: PowerShell Remoting (WinRM)

```
$session = New-PSSession -ComputerName node-01 -Credential $cred
```

Result: connection refused. The firewall blocked port 5985 entirely. Not from the subnet, not from anywhere. WinRM on these nodes was simply off.

I wasn't surprised. Azure Local documentation recommends keeping WinRM limited or disabled for security. I'd hoped the cluster management layer would expose it internally. It didn't.

Attempt 2: Azure Arc Run Commands

Arc Run Commands is an Azure feature that lets you execute scripts on Arc-connected machines through the Azure control plane — no direct network connection needed. It looked perfect. The REST API accepted the command. The Azure portal showed it as "Creating." The expected outputs: exit code 0, script output, completion timestamp.

Actual output: exit code 0, no script output, no error.

The commands were being accepted by Azure, acknowledged as successful, and then... nothing. Or they were running silently and returning nothing. There was no way to distinguish between "command ran and produced no output" and "command never ran." We tried six variations. Different encoding. Different REST API versions. Different script content. Always the same result: accepted, acknowledged, silent.

This failure mode — silent success that wasn't actually success — was worse than an error. With an error, you know something is wrong. With a silent exit-code-0, you start questioning whether your expectations were wrong. You spend 45 minutes investigating a thing that was never going to work.

This is the first rule that emerged from the project, though I didn't write it down yet:

> ***When an approach has failed three times with different variations, the approach is wrong — not the execution.***

Attempt 3: Azure Arc SSH

The pivot came from stepping back and asking a different question: what is Azure Arc *for*? Arc exists to give you a management plane over connected machines. SSH access is one of the things Arc is designed to enable. The documentation described `az ssh arc` — an Azure CLI command that tunnels

an SSH connection through the Arc agent, through Azure's backbone, to the target machine. No VPN needed. No firewall changes needed. The tunnel routes through the same channel the Arc agent uses to phone home to Azure.

The Arc agent was already running on these nodes. Azure Arc was already managing them. The path was already there. We just hadn't looked at it.

```
az ssh arc \
  --resource-group <cluster-resource-group> \
  --name node-01 \
  --local-user svc.AzureLocal \
  -- -o StrictHostKeyChecking=no "powershell -Command whoami"
```

Result:

```
yourdomain\svc.azurelocal
```

It worked. The credentials were valid. Node-01 was accessible.

That's the moment I understood what Arc SSH actually is: not a workaround, but the designed path. When everything else on Azure-connected machines fails — WinRM blocked, VPN nonexistent, RDP not configured — Arc SSH is usually the answer. Because Arc is already there. It's already connected. You're just using the connection that already exists.

The Decision Architecture

Before we went further, I made a decision about how Claude and I would work together through the rest of the project. Not explicitly at first — it crystallized over the first two sessions and I wrote it down later — but the structure was already forming.

I would carry: Which node to start with and why. Whether a step was safe to run. Business context (this is a production cluster, the local operations team is offline, if this breaks there's nobody local to fix it). Domain knowledge about the infrastructure topology. Decision authority for anything irreversible.

Claude would carry: PowerShell command syntax. Azure CLI command construction. API request formatting. Error pattern recognition. Knowing when a PowerShell cmdlet requires CredSSP and the Arc SSH session doesn't support it.

This isn't a restriction on what AI can do. It's a recognition of what each party is actually good at. I'm not going to out-iterate Claude on PowerShell syntax. Claude is not going to out-judge me on what happens to the operations team if node-01 goes offline unexpectedly.

Neither side is sufficient without the other. Claude could generate every command correctly and still make the wrong call about when to run it. I could make every right call about sequencing and still spend an hour wrestling with escape character syntax in a nested PowerShell-over-Arc-SSH command chain. The division of labor isn't philosophical. It's just practical.

RULES WRITTEN: CHAPTER 1

Rule C1-1 — Arc SSH First: On Azure-connected machines with no conventional remote access, try `az ssh arc` before any other method. Arc is already managing the machine; use the channel that's already there.

Rule C1-2 — Scaffold, Don't Trust: Training data is a starting scaffold, not authoritative documentation. Build the plan from training data. Validate every assumption against live sources before executing.

Rule C1-3 — Credentials Are Links, Not Values: Store secrets in Key Vault. Give the AI Key Vault secret names, not the secret values. The AI generates commands that pull from Key Vault at runtime. The AI never sees the actual credential.

Rule C1-4 — Three Failures, Stop: If the same approach has failed three times with different variations, the approach is wrong. Stop. State what you tried and why it didn't work. Ask for a different direction.

CHAPTER 2

The Lockout

How an AI Broke Production, Twice, in the Same Session

I'm going to tell you this story in chronological order because the timeline matters. The decisions at each moment were reasonable given what was known at that moment. That's what makes this instructive rather than just embarrassing.

The session had been going well. We'd established Arc SSH access. Credentials were validated. The WDAC module was confirmed present. The deployment plan was ready to execute. All that remained was a few setup steps — configure the SSH daemon for security, establish key-based auth so future sessions wouldn't require interactive password entry — and then begin the actual agent deployment.

T+0: The Setup Script

Claude generated `setup-node01.ps1`. The script was reasonable. It configured a few sshd settings, established key-based auth, set up some firewall rules.

Among its contents: `ListenAddress 127.0.0.1`.

The intent: restrict the SSH daemon to listen only on localhost. This is standard security hardening advice. Restrict attack surface. Don't expose sshd on all interfaces if you only need it locally.

The problem: Azure Arc SSH does not work the way most people assume.

When you run `az ssh arc`, you're not opening a direct TCP connection from your machine to port 22 on the target node. The connection routes through Azure's control plane: your client → Azure Resource Bridge → Arc agent on

the node → Arc proxy → sshd. The Arc proxy, running locally on the node, connects to sshd through **IPv6 loopback — `[::1]:22` — not IPv4 `127.0.0.1:22`.

`ListenAddress 127.0.0.1` tells sshd: listen on IPv4 localhost only.

The Arc proxy connects via IPv6.

These two facts are incompatible.

T+5: The Session Drops

The script ran. sshd restarted. The Arc SSH session dropped — expected behavior, sshd was restarting. I tried to reconnect.

> `HTTP 503 — Service Unavailable`

sshd was now listening on `127.0.0.1:22` only. The Arc proxy was connecting to `[::1]:22`. The IPv6 loopback was refused. The connection was dead.

Node-01 was completely inaccessible.

Let me describe what that actually means in this environment. No VPN. No RDP. No WinRM from outside the subnet. No iDRAC. No KVM-over-IP. The node is in a remote data center. The on-site team was unavailable. Arc SSH was **the only way in**, and the AI had just broken it.

T+10 Through T+60: The Flailing

What followed was approximately 60 minutes of escalating attempts, each one building on a foundation of broken assumptions.

Claude tried Arc Run Commands to fix the sshd_config on node-01. Arc Run Commands had already been documented as unreliable on these nodes earlier the same day. They still didn't work. The commands were accepted, acknowledged, and then silently did nothing.

Claude tried targeting sibling nodes. The other three nodes had Arc agents but hadn't had `azcmagent config set incomingconnections.ports 22` run on them yet. SSH wasn't enabled. They returned 404.

Claude tried Arc Run Commands on the sibling nodes to enable SSH. Same silent failures.

Claude tried extension reinstallation. Uncertain whether it would actually reset the sshd_config.

Each attempt was technically sophisticated. The variation between attempts was real. But the underlying approach — fix node-01 through Azure's control plane without direct access — was broken, and no variation of the execution was going to fix that.

This is what I mean when I say "flailing." The AI doesn't feel the weight of "it's been an hour and we're no closer." It doesn't have the instinct to stop and say "wait, we're iterating on a broken approach." It just tries the next variation, because that's what the problem-solving loop does. That's not a criticism — it's a description of how the execution engine works. The judgment that the approach itself is wrong has to come from somewhere else.

T+90: The Human Judgment Call

I stepped back and thought about what I actually knew about this cluster.

The four nodes were on the same subnet. They were cluster members. Windows clusters have well-known trust relationships between members. WinRM was blocked at the external firewall, but cluster members typically trust each other at the network layer. The question was: does WinRM between node-04 and node-01 work, even if WinRM from the outside world doesn't?

I had one other path in: an earlier portal session on node-04 with interactive password auth through the Azure portal's Cloud Shell. That session was still technically active, though barely.

From node-04:

```
# No -Credential flag — nodes trust each other via Kerberos on the
same subnet
# Adding -Credential explicitly causes Access Denied (double-hop
constraint)
Invoke-Command -ComputerName node-01 -ScriptBlock {
    $c = Get-Content 'C:\ProgramData\ssh\sshd_config'
    $c = $c | Where-Object { $_ -ne 'ListenAddress 127.0.0.1' }
    Set-Content 'C:\ProgramData\ssh\sshd_config' $c
    Restart-Service sshd
}
```

Result: it worked. WinRM lateral movement from node-04 to node-01, over the cluster subnet, with no credentials needed.

Arc SSH was restored.

The `-Credential` detail is worth noting because it's counterintuitive. The instinct when making a remote call to another machine is to pass credentials. On HCI cluster nodes, this breaks the authentication. The nodes trust each other via Kerberos passthrough when you're already authenticated as a domain account. Specifying `-Credential` forces a new authentication handshake that fails due to the Kerberos double-hop constraint. The fix is to do less, not more. This became a documented gotcha.

The judgment call that found the recovery path — "WinRM between cluster nodes probably works because they're on the same subnet" — came from experience with how Windows cluster trust relationships work at the network layer. That fact was not in Claude's training data. It's not in most SSH guides. It comes from having set up clusters before and knowing how these things work.

That's the trust boundary made visible. Claude had been iterating on variations of a broken approach for 60 minutes. I found the recovery in 30 minutes, not because I was iterating faster but because I knew something the AI didn't: infrastructure topology knowledge specific to how HCI clusters behave.

The Second Lockout (Same Session)

Node-01 was recovered. The immediate priority: establish proper key-based auth so future sessions didn't require interactive password entry.

Claude proposed deploying an SSH public key.

The key landed in `C:\Users\svc.AzureLocal\.ssh\authorized_keys`.

This is correct on Linux. It's correct for standard user accounts on Windows OpenSSH. It is wrong for administrator accounts on Windows OpenSSH.

On Windows, the OpenSSH server handles administrator accounts differently. When the connecting user is a member of the Administrators group, sshd looks for authorized keys in a shared file: `C:\ProgramData\ssh\administrators_authorized_keys`. Not the user's home directory. This is a Windows-specific behavior, documented in Microsoft's OpenSSH documentation but not prominently, and missing entirely from most SSH guides that were written for Linux environments.

The wrong key location alone would just have meant key auth didn't work. What made it catastrophic was the ACL.

Claude set access control on the wrongly-placed file with permissions that sshd interpreted as insecure. Windows OpenSSH is strict about file permissions on authorized_keys files. If the ACL doesn't meet sshd's expectations, sshd doesn't just skip key auth — it rejects the authorization check entirely. Before even attempting a password prompt.

All remote access to node-01 was broken. Again. Same session.

Recovery was the same path: WinRM lateral from node-04. Delete the bad file. Place the key correctly in `C:\ProgramData\ssh\administrators_authorized_keys`. Set the correct ACL: `SYSTEM:(F)` and `BUILTIN\Administrators:(F)` only, inheritance removed.

```
# From node-04, fix the key location and ACL on node-01
Invoke-Command -ComputerName node-01 -ScriptBlock {
    # Remove the wrong file
    Remove-Item 'C:\Users\svc.AzureLocal\.ssh\authorized_keys' -
Force -ErrorAction SilentlyContinue

    # Write the key to the correct location
    [System.IO.File]::WriteAllText(
        "C:\ProgramData\ssh\administrators_authorized_keys",
        "ssh-ed25519 AAAA...your-public-key-here... arc-ssh-key"
    )

    # Set strict ACL — inheritance removed, only SYSTEM and
Administrators
    icacls "C:\ProgramData\ssh\administrators_authorized_keys" /
inheritance:r /grant "SYSTEM:(F)" /grant "BUILTIN\Administrators:
(F)"
}
```

That `[System.IO.File]::WriteAllText()` instead of `Set-Content` matters. When delivering text through a WinRM session, `Set-Content` wraps long strings based on the remote console width. An SSH public key is typically 400+ characters on a single line. `Set-Content` through WinRM splits it across two lines. An SSH key split across two lines is an invalid SSH key. `WriteAllText` bypasses this — it writes the exact string as a single atomic file operation, console width be damned.

Third incident, same session: creating the backup admin account with `net user AccountName <24-character-password> /add` produced an interactive "Do you want to continue? [Y/N]" prompt. The non-interactive Arc SSH session hung. Fix: `New-LocalUser` with `ConvertTo-SecureString` — fully non-interactive, no length limits, no prompts.

What I Felt at T+60

I need to say this plainly, because the narrative arc — broke it, fixed it, learned — makes it sound cleaner than it was.

At T+60, I had been locked out of a production node for an hour. The node was in a remote data center. The on-site team was unavailable. There was no physical access. There was no out-of-band management. The only remote access method the entire project depended on was the one the AI had broken. And the AI was still generating variations of a command that had failed six times.

The feeling wasn't anger. It was something colder: the recognition that you are alone with a broken system, thousands of miles away, and the tool you've been collaborating with doesn't understand that its approach isn't working. It can't feel the weight of "this is production infrastructure." It just tries the next variation.

That's not a failure of AI. That's a description of what AI is and isn't. The AI executes brilliantly. The judgment about when the execution strategy is wrong has to come from a human.

The recovery came from infrastructure topology knowledge — understanding that cluster nodes on the same subnet trust each other via WinRM. That knowledge wasn't derivable from any command. It came from experience.

This is the moment the trust boundary model became explicit in my head. The human carries domain judgment. The AI carries execution. When those roles get confused — when the AI is in the "judgment" lane — that's where production breaks.

RULES WRITTEN: CHAPTER 2

Rule C2-1 — Never Modify sshd_config: Do not rewrite, replace, or remove lines from `C:\ProgramData\ssh\sshd_config` on any cluster node. If Arc SSH breaks, the problem is almost never sshd_config. The one time it was sshd_config, an AI caused a production lockout.

Rule C2-2 — No ListenAddress Changes: Do not set `ListenAddress` in sshd config. Arc SSH requires sshd to listen on IPv6 loopback `[::1]:22`. Any `ListenAddress` directive that excludes IPv6 breaks Arc SSH.

Rule C2-3 — Admin SSH Keys Go in administrators_authorized_keys: On Windows OpenSSH, administrator accounts use `C:\ProgramData\ssh\administrators_authorized_keys`, not `~\.ssh\authorized_keys`. ACL must be `SYSTEM:(F)` and `BUILTIN\Administrators:(F)` only — inheritance removed.

Rule C2-4 — WriteAllText, Not Set-Content: When deploying SSH keys or config content via WinRM, use `[System.IO.File]::WriteAllText()`. `Set-Content` wraps long strings on the remote console width and will corrupt SSH keys.

Rule C2-5 — New-LocalUser, Not net user: Creating local accounts non-interactively requires `New-LocalUser` with `ConvertTo-SecureString`. `net user` prompts interactively for passwords over 14 characters.

Rule C2-6 — WinRM Between Cluster Nodes Needs No Credentials: `Invoke-Command -ComputerName <node>` between cluster members works via Kerberos passthrough. Do not add `-Credential`. Adding `-Credential` explicitly causes Access Denied.

Rule C2-7 — Security Hardening Deferred: Do not apply security hardening during initial setup. Get it working. Verify end-to-end. Then, with explicit approval and a stated rollback plan, consider hardening.

Rule C2-8 — Stop Flailing: After three failed attempts with the same approach, the approach is wrong. Stop. State what was tried and why it failed. Explicitly ask for a different direction before continuing.

CHAPTER 3

The Rules

The Governance Framework That Came Out of the Wreckage

The morning after the lockout session, I sat down and wrote rules.

Not "lessons learned" in the corporate retrospective sense — vague aspirations about process improvement that get filed away and ignored. Actual rules, in a file called `CLAUDE.md` that lives in the project directory and gets loaded into Claude's context every single time a session opens on this project.

The distinction matters. A lesson learned is information. A rule encoded in tooling is a constraint. Information can be forgotten, deprioritized, or overlooked under pressure. A constraint in the session context cannot be ignored — Claude reads it before every response.

How CLAUDE.md Works

Every Claude Code project can have a `CLAUDE.md` in its root directory. Claude reads this file at the start of every session. It's not a suggestion — it's part of the session context. Instructions in CLAUDE.md override default behavior.

There's also a global version at `~/.claude/CLAUDE.md` — loaded for every project, on every machine. The project-level CLAUDE.md extends the global one with project-specific rules.

The global file got eight core rules after the lockout. The project file got sixteen more. Every rule traces to a specific incident with a specific root cause. Not "be careful with sshd_config" — but "do not modify `C:\ProgramData\ssh\sshd_config` because an AI session modified it on 2026-03-13 and set `ListenAddress 127.0.0.1`, which broke Arc SSH because Arc's proxy connects via IPv6 loopback `[::1]:22`, causing a complete production lockout."

The specificity is the point. Vague rules invite interpretation. Specific rules don't.

Claude Code Hooks as Circuit Breakers

Claude Code supports hooks — shell-level interceptors that catch specific command patterns before they execute and either block them entirely or pause for human confirmation.

CLAUDE.md rules work at the prompt level. Claude reads the rules and follows them — until, hypothetically, context pressure, model behavior drift, or an unusual chain of reasoning leads to a response that violates a rule.

Hook rules go one level deeper: they intercept tool calls before they execute and block them at the shell level, before Claude even returns a response.

The project uses several hook rules:

```
block-sshd-config      — blocks any command referencing sshd_config
block-listenaddress    — blocks any command containing ListenAddress
block-firewall-changes — blocks New-NetFirewallRule and netsh
advfirewall
block-winrm-enable     — blocks Enable-PSRemoting and winrm
quickconfig
block-danger-script    — blocks specific dangerous scripts by
filename
```

These aren't warnings. They're hard stops. The command doesn't execute. The session pauses. The user sees what was blocked and why.

There are also warn rules — operations that pause and require explicit confirmation before proceeding:

```
warn-sshd-restart     — Restart-Service sshd (if sshd dies, all
access is lost)
warn-service-changes  — Set-Service, Stop-Service (need a rollback
plan stated first)
warn-registry-changes — HKLM\CurrentControlSet\Services writes
warn-multi-node       — commands referencing nodes 02-04 (rollout
gate enforcement)
```

The hook rules for firewall changes are interesting to think about. There's a legitimate step in the deployment where I need to open firewall ports for the management appliance subnet. The hook rule blocks `New-NetFirewallRule` by default. So the procedure for that step is:

1. Open the hook rule configuration file
2. Set `enabled: false` for the firewall block rule
3. Run the specific approved firewall command
4. Immediately set `enabled: true` again
5. Document what was done in the session checkpoint

This is a circuit breaker. The rule isn't "never do this" — it's "always do this consciously, with a named action and a deliberate re-enable." The overhead is intentional. It creates a moment of friction that forces explicit human approval in the exact moment where automation would otherwise glide past it.

The Trust Boundary, Formalized

The rules that came out of the lockout incident aren't just a list of things not to do. They encode a model of how human and AI judgment should interact.

Here's what emerged from watching the lockout session:

The AI is in the wrong lane when it:

- Decides whether a security hardening step is appropriate for a production environment

- Continues iterating on a broken approach without flagging that the approach may be wrong
- Applies general best practices without verifying they apply to the specific platform
- Makes irreversible changes without stating the rollback path first

The human is in the wrong lane when they:

- Try to remember the exact `icacls` syntax for removing ACL inheritance
- Try to mentally translate a sequence of hex error codes from a PowerShell exception
- Try to write a base64-encoded binary transfer without a tool to check the encoding
- Try to track which of six similar commands they've already tried

The trust boundary is asymmetric because the failure modes are asymmetric. This is why the rules skew toward constraining AI judgment on irreversible operations rather than constraining AI execution on technical tasks.

Rules Compound; They Don't Just Accumulate

After the lockout, I had rules about sshd_config. Those rules prevented the same class of failure in every subsequent session. But more importantly: they informed how I thought about related problems.

The rule "no ListenAddress changes" created the habit of verifying IPv4/IPv6 assumptions before any sshd modification. That habit prevented a hypothetical future lockout on a different node from a different AI error pattern.

The same pattern repeated across WinRM string handling and SSH behavior differences — caution from one rule preventing errors in completely unrelated operations.

The CLAUDE.md as Operational Manual

By the end of the project, the CLAUDE.md for this project had sixteen specific rules, each with a named root cause. The global `~/.claude/CLAUDE.md` had eight more that applied to every project.

It also had:

- The complete proven deployment procedure, with expected outputs for each step
- The node state tracking table (which nodes have which steps complete)
- The Key Vault layout (which vault has which secrets)
- The API query patterns that were validated (and the ones that were invalid)
- The known failure modes and how to diagnose them

When a new session opens on this project, Claude has access to months of hard-won operational knowledge from the first load of the context. Not from memory — from a file. That distinction matters. Memory isn't reliable at session boundaries. Files are.

This is what institutional memory looks like for a one-human, one-AI team. You write down what you learned in the session that taught you. You put it somewhere Claude will see it next time. You make the rules specific enough that they can't be misinterpreted.

The methodology didn't exist before the project. It emerged from the project. Every incident produced an artifact. Every artifact made the next session safer.

RULES WRITTEN: CHAPTER 3

Rule C3-1 — Rules Go in CLAUDE.md, Not in Memory: Operational rules must be written to a file that Claude reads at session start. Memory doesn't survive session boundaries. Files do.

Rule C3-2 — Specific Rules Only: Rules must trace to a specific root cause with a specific date. "Be careful with sshd" is not a rule. "Do not set ListenAddress because on 2026-03-13 this broke Arc SSH's IPv6 proxy" is a rule.

Rule C3-3 — Encode Rules in Tooling: Critical rules should be enforced by hook rules, not just written in CLAUDE.md. A block rule can't be overridden by context pressure. A document rule can.

Rule C3-4 — Document in the Session That Teaches: Write the rule in the session that produced the knowledge. "We should document this" is a promise that doesn't get kept. The RCA written at 2am is still there at 9am.

Rule C3-5 — No Irreversible Action Without a Rollback Plan: Before any command that changes a service state, access configuration, or security policy: state what it does, what breaks if it fails, and the exact command to undo it.

CHAPTER 4

The Deployment

Enterprise Software on WDAC-Enforced Azure Local, All Four Nodes, Working

By session three, the access problem was solved. Non-interactive Arc SSH key auth was working on all four nodes. Credentials were in Key Vault. A backup local admin account (`ArcAdmin`) existed on each node, also with key auth, in case the domain account had problems. The session setup was:

```
az ssh arc \
  --resource-group <cluster-rg> \
  --name node-01 \
  --local-user svc.AzureLocal \
  --private-key-file ./keys/ssh-key \
  -- -o StrictHostKeyChecking=no -o UserKnownHostsFile=/dev/null \
  "powershell -Command <command>"
```

Everything before this had been getting to the starting line. Now the actual mission could begin.

The WDAC Problem

Windows Defender Application Control in Enforced mode maintains an allowlist of what can run. On Azure Local, this is on by default and managed by the cluster infrastructure. The software's binaries aren't on the allowlist. So when the agent tries to run — even after successful installation — WDAC blocks the processes.

The solution: a supplemental policy. WDAC supports policy layering — you can add a supplemental policy that extends the base policy without replacing it. In this case, a FilePath-based supplemental policy that allowlists the specific directories where the agent installs its binaries.

This is the document I'd obtained through the vendor's support channel. It hadn't been published publicly. The XML defined 16 FilePath rules covering the directories the agent uses, plus 2 hash rules for specific binaries. Policy ID: fixed in the file, non-negotiable.

The 11-Step Procedure (What Nobody Documented)

The procedure that emerged from the deployment sessions:

Step 1: Verify WDAC cmdlets are available

```
Get-Command *AsWdac* -ErrorAction SilentlyContinue
```

Expected: `Enable-ASLocalWDACPolicy`, `Get-ASLocalWDACPolicyMode`, `Add-ASWDACSupplementalPolicy` from module `Microsoft.AS.Infra.Security.WDAC`. If these aren't present, the node doesn't have the Azure Local WDAC management module. Stop and investigate.

Step 2: Confirm WDAC is in Enforced mode

```
Get-ASLocalWDACPolicyMode
```

Expected: `2` (Enforced). Use `Get-ASLocalWDACPolicyMode`, not `Get-ASWDACPolicyMode`. The per-node cmdlet works via Arc SSH; the cluster-wide cmdlet requires CredSSP, which Arc SSH doesn't support. The cluster-wide cmdlet will fail silently or error in ways that obscure the root cause.

Step 3: Switch to Audit Mode (this node only)

```
Enable-ASLocalWDACPolicy -Mode Audit
Get-ASLocalWDACPolicyMode
```

Expected: `1` (Audit). This is temporary — we're in Audit mode only long enough to install the agent. Same cmdlet rule applies: `Enable-ASLocalWDACPolicy`, not `Enable-ASWDACPolicy`.

Step 4: Download the agent installer from the management appliance

```
$null = New-Item -ItemType Directory -Force -Path 'C:
\tmp\DPAgentInstall'
Invoke-WebRequest -Uri 'https://dp-mgmt-appliance/connector/
DPAgent.zip' `
    -OutFile 'C:\tmp\DPAgentInstall\DPAgent.zip' -UseBasicParsing
Expand-Archive -Path 'C:\tmp\DPAgentInstall\DPAgent.zip' `
    -DestinationPath 'C:\tmp\DPAgentInstall' -Force
```

Have the node pull the installer directly from the management appliance, not through the Arc SSH tunnel. Arc SSH tunnel throughput is approximately 40KB/s — a 31MB installer takes 13 minutes through the tunnel versus under a minute from inside the local network. Do not pipe large files through Arc SSH.

(`dp-mgmt-appliance` and `DPAgent.zip` are fictional stand-ins. Substitute your management appliance hostname and agent package name.)

Step 5: Transfer the WDAC policy XML

The policy XML can't be pulled from a URL — it lives on your local machine. It needs to go to the node. Arc SSH supports this but with a constraint: command length limits. The XML is too large to encode inline in a single command.

Solution: base64 in two chunks.

```
# Split the XML at a safe boundary
PART1=$(head -c 3168 policy/Supplemental_Policy.xml | base64 -w0)
PART2=$(tail -c +3169 policy/Supplemental_Policy.xml | base64 -w0)

# Write part 1
az ssh arc ... "powershell -Command \"\
$b=[System.Convert]::FromBase64String('$PART1'); \
[System.IO.File]::WriteAllBytes('C:\\tmp\\Supplemental_Policy.xml',\
$b)\""

# Append part 2 and validate XML
az ssh arc ... "powershell -Command \"\
$b1=[System.IO.File]::ReadAllBytes( \
'C:\\tmp\\Supplemental_Policy.xml'); \
\$b2=[System.Convert]::FromBase64String('$PART2'); \
\$c=\$b1+\$b2; \
[System.IO.File]::WriteAllBytes('C:\\tmp\\Supplemental_Policy.xml',\
$c); \
[xml](Get-Content 'C:\\tmp\\Supplemental_Policy.xml')|Out-Null; \
Write-Output 'XML valid'\""
```

That `[xml](...)|Out-Null` at the end: this validates that the XML is well-formed. If the transfer corrupted the file (truncation, encoding issue, wrong split point), this throws before you try to deploy the policy with the corrupted file and get a cryptic error code from `citool`.

The split point of 3168 bytes is not magic — it's just under the Arc SSH command length limit with encoding overhead. If your XML file differs significantly in size, adjust the split point and potentially use more chunks.

Step 6: Install the agent MSI silently

```
Start-Process msiexec -ArgumentList '/i','C:
\tmp\DPAgentInstall\DPAgent.msi','/qn','/L*V', \
'C:\tmp\DPAgent_install.log' -Wait
Start-Sleep 5
Get-Service 'DP Agent' -ErrorAction SilentlyContinue
```

Expected: `Status: Running`. If the service isn't running, check the install log.

Step 7: Deploy the supplemental WDAC policy

This is where the first major gotcha lived.

The documented cmdlet for deploying a supplemental policy is `Add-ASWDACSupplementalPolicy`. That cmdlet requires CredSSP. Arc SSH doesn't support CredSSP. The call fails with an access denied or hangs indefinitely.

The fix: go lower. `citool` is the underlying Windows App Control management tool. It doesn't require CredSSP. It takes a compiled binary policy file (`.cip`), not the XML.

```
# Compile the XML to binary
ConvertFrom-CIPolicy -XmlFilePath 'C:\tmp\Supplemental_Policy.xml' \
-BinaryFilePath 'C:\tmp\Supplemental_Policy.cip'

# Deploy the compiled policy
citool --update-policy 'C:\tmp\Supplemental_Policy.cip' --json
```

Expected: `OperationResult: 0`. Any other result means the policy didn't deploy. Check that the XML was valid (Step 5 validation) and that the binary compiled correctly.

The `citool` path for WDAC policy deployment works without CredSSP on Windows 10.0.26100 and later. On older Windows versions, verify that `citool` is available and supports the `--update-policy` flag.

Step 8: Switch back to Enforced mode

```
Enable-ASLocalWDACPolicy -Mode Enforced
Get-ASLocalWDACPolicyMode
```

Expected: `2`. WDAC is back in Enforced mode with the supplemental policy active.

Step 9: Restart the agent service and verify both processes

```
Restart-Service 'DP Agent'
Start-Sleep 8
Get-Process -Name dpa,dpb -ErrorAction SilentlyContinue
```

Expected: both `dpa` and `dpb` listed in the process table — this is the WDAC success signal. If WDAC is blocking the binaries, the service restarts but the processes don't appear. If both processes are listed, WDAC is allowing them. *(`dpa` and `dpb` are fictional stand-ins; substitute your agent's actual process names.)*

Step 10: Open firewall ports

The management appliance needs to connect to the node on the agent's required ports (substitute your software's required ports — this example uses 12800 and 12801). The firewall rule needs to cover the management appliance's full address range — not just the VIP/DNS address, but all per-node appliance IPs.

Management appliance clusters use multiple IPs: one VIP for DNS resolution, and individual node IPs for actual connections. A firewall rule targeting only the VIP will fail when the appliance's per-node IPs make the connection. Use the subnet covering all appliance IPs:

```
# Disable the hook rule block-firewall-changes first
# Re-enable immediately after

New-NetFirewallRule -DisplayName 'DP Agent Port 12800' -Direction
Inbound \
-Protocol TCP -LocalPort 12800 -RemoteAddress '10.x.y.0/24' -Action
Allow -Profile Any
New-NetFirewallRule -DisplayName 'DP Agent Port 12801' -Direction
Inbound \
-Protocol TCP -LocalPort 12801 -RemoteAddress '10.x.y.0/24' -Action
Allow -Profile Any
```

Both ports. Both rules. Both required.

Step 11: Register the host with the management appliance

```
# Get OAuth token using host-registration service account
$creds = Get-Content "host-registration-creds.json" | ConvertFrom-
Json
$tokenBody = @{
    client_id = $creds.client_id
    client_secret = $creds.client_secret
} | ConvertTo-Json
$TOKEN = (Invoke-RestMethod -Method POST -Uri
$creds.access_token_uri `
    -Headers @{'Content-Type'='application/json'} -Body
$tokenBody).access_token

# Register the host
$registrationBody = @{
    query = '# your vendor host registration mutation here'
    variables = @{
        input = @{
            clusterUuid = 'your-cluster-uuid'
            hosts = @(@{ hostname = 'node-01'; hasAgent = $true })
        }
    }
} | ConvertTo-Json -Depth 10

Invoke-RestMethod -Method POST `
    -Uri 'https://your-vendor-api-endpoint/api/graphql' `
    -Headers @{'Authorization'="Bearer $TOKEN"; 'Content-
Type'='application/json'} `
    -Body $registrationBody
```

If you get a timeout error from the vendor API: the firewall ports aren't open, or you used the wrong management appliance IPs (VIP-only rule instead of /24 subnet).

The Compounding Story

By the time we reached the deployment, the first session's knowledge base had become something different. It had started as training data — a scaffold. Each subsequent session had filled gaps, corrected errors, and added knowledge grounded in real execution.

Session 1: compiled knowledge base from training data. No live system access. Provisional.

Session 2: three access method failures. One lockout. Four incidents. Eight rules. Non-interactive key auth on all four nodes by end of session.

Session 3: WDAC procedure validated on node-01. `citool` substituted for `Add-ASWDACSupplementalPolicy`. Base64 chunking technique developed for XML transfer. Installer pull-from-management-appliance technique validated.

Session 4-6: repeated procedure on nodes 02, 03, 04. Each node took under an hour. No new incidents. The rules from sessions 1-3 prevented the failure modes that session 2 discovered.

Final state: all four nodes connected and upgraded in the vendor's management console. Both agent processes running. Firewall ports confirmed open. Restore testing underway.

The distance between session 1 and session 8 isn't eight sessions of effort. It's eight sessions of compounding. Each session was faster than the last, not because the work got easier, but because the accumulated rules and procedures made it harder to make mistakes.

The End State

Node	Arc SSH	WDAC Policy	Agent	Firewall	Console
01	✓	✓ Deployed	CONNECTED	✓ Ports OK	✓ REGISTERED
02	✓	✓ Deployed	CONNECTED	✓ Ports OK	✓ REGISTERED
03	✓	✓ Deployed	CONNECTED	✓ Ports OK	✓ REGISTERED
04	✓	✓ Deployed	CONNECTED	✓ Ports OK	✓ REGISTERED

The Division of Labor

Let me be specific about the division of labor in the deployment phase, because this is where the trust boundary model is most visible.

Claude carried:

- Generating the correct Arc SSH command structure with proper escaping for PowerShell nested inside Bash inside Arc SSH
- Calculating the correct base64 split point for the XML chunked transfer
- Translating the decimal error code `-2147024894` from `citool` into "this is `ERROR_FILE_NOT_FOUND` — the XML path is wrong or the file is corrupted"
- Recognizing that `Add-ASWDACSupplementalPolicy` requires CredSSP and proposing `citool` as the alternative
- Writing the vendor's API mutations for host registration from schema knowledge
- Constructing the OAuth2 token request body correctly

I carried:

- Which node to start with and why (node-01, isolated from the others, so a failure wouldn't affect production VMs running on nodes 02-04)
- When to proceed vs when to pause and verify (every WDAC mode transition: I confirmed the mode before and after)
- The decision to use the /24 subnet for the firewall rule (I knew the management appliance cluster topology from the infrastructure setup)
- When the hook rule for firewall changes needed to be disabled and the specific approved operation to allow
- The judgment call on when to count a node as "done" (waiting for both agent processes, not just the service status)

Neither side was sufficient. The deployment wouldn't have happened without Claude generating correct PowerShell across eleven steps times four nodes. Claude wouldn't have known which node to target first, or when to wait, or which subnet to use. That's not a limitation. That's the design.

RULES WRITTEN: CHAPTER 4

Rule C4-1 — Per-Node WDAC Cmdlets Only: Use `Get-ASLocalWDACPolicyMode` and `Enable-ASLocalWDACPolicy` (per-node). Never `Get-ASWDACPolicyMode` or `Enable-ASWDACPolicy` (cluster-wide, requires CredSSP).

Rule C4-2 — citool, Not Add-ASWDACSupplementalPolicy: `Add-ASWDACSupplementalPolicy` requires CredSSP. Arc SSH doesn't support CredSSP. Use `ConvertFrom-CIPolicy` to compile XML to binary, then `citool --update-policy` to deploy.

Rule C4-3 — Nodes Pull Large Files from Local Sources: Do not pipe files larger than ~1MB through the Arc SSH tunnel. Have the node pull from internal management appliance or other local network sources. Tunnel throughput (~40KB/s) makes large transfers unacceptably slow.

Rule C4-4 — Validate XML Before Deploying: After base64 chunked transfer, validate the XML with `[xml](Get-Content 'path.xml') | Out-Null` before attempting policy deployment. A corrupt XML produces a cryptic error from `citool` that obscures the real cause.

Rule C4-5 — Management Appliance Firewall Rule Needs /24 Subnet: Management appliance clusters connect from per-node IPs, not just the VIP. DNS resolves to the VIP, but actual traffic comes from individual management appliance node IPs. Firewall rules targeting only the VIP will timeout. Use the /24 subnet covering all management appliance IPs.

Rule C4-6 — Both Ports Required: The agent requires both listening ports — ports 12800 and 12801 in this example (substitute your software's actual ports). A rule opening only one port will cause registration timeout. The health signal is both agent processes listed in the process table.

Rule C4-7 — Separate Host-Registration and Monitoring Credentials: The vendor's host registration API call requires the host-registration service account, not the main monitoring API credentials. They're different service accounts with different scopes.

What Comes Next

The project moved into 3rd Party software testing after all four nodes were connected — taking baseline snapshots of a test VM, enrolling additional services, and observing whether the configurations survived operational changes. That's the next story.

This is Book 1 in the Knowledge Gap: Detonator series.

Next up: *Knowledge Gap: Detonator — Claude Code Managing Azure Local: V2 Deploying VMs from Marketplace Using Claude Code*

About the Author

Dustin Winkler is the founder of AWACS LLC. He builds AI-assisted operational tooling for infrastructure teams. By day he manages Azure infrastructure at enterprise scale. The rest of the time he documents what actually happens when AI meets production. This is the first thing he's published. It won't be the last.

Let's Connect

Website: awacs.ai
GitHub: github.com/Dwink213
LinkedIn: linkedin.com/in/dustin-winkler-nc

Ready to Use Claude Code in Your Infrastructure Operations?

Start with the rules in Chapter 3. Write them before you need them.

Need someone to build this for your team? Book time and let's talk about what your version looks like.

Book a call: calendly.com/dustin-dustinwinkler/30min

Published by AWACS LLC — awacs.ai

Employer-specific infrastructure details, vendor names, node identifiers, IP addresses, and identifying information have been generalized or removed. The technical procedures are based on real validated sessions and apply to any Azure Local / Azure Stack HCI environment deploying enterprise third-party software.

KNOWLEDGE GAP

DETONATOR

$9.99

> *"The AI locked me out of a production node 4,000 miles away. There was no VPN, no RDP, no out-of-band management. Arc SSH was the only way in, and the AI had just broken it."*

This is the field journal from a real production deployment. No theory. No best practices guide. Just what happened when an infrastructure engineer used Claude Code to deploy enterprise software on WDAC-enforced Azure Local nodes, got locked out twice in the same session, and wrote the governance framework that came out of the wreckage.

INSIDE

01 The full incident timeline from zero access to production lockout to lateral recovery

02 The CLAUDE.md governance framework and hook circuit breakers that prevent the same class of failure from recurring

03 The trust boundary model: where human judgment ends and AI execution begins

04 The complete deployment procedure for enterprise third-party software on WDAC-enforced Azure Local

ABOUT THE AUTHOR

Dustin Winkler is the founder of AWACS LLC. He builds AI-assisted operational tooling for infrastructure teams. By day he manages Azure infrastructure at enterprise scale. The rest of the time he documents what actually happens when AI meets production. This is the first thing he's published. It won't be the last.