Integrating Ollama with Deep System Capabilities on Windows via PowerShell

1. Introduction

1.1. Project Goal and Scope

This report details the creation and configuration of a sophisticated environment on Microsoft Windows, leveraging PowerShell scripting to deeply integrate the Ollama local Large Language Model (LLM) hosting platform. The objective is to establish an Ollama instance endowed with significant operational capabilities, including live web search, comprehensive filesystem access, command execution (PowerShell and CMD), interaction with the host operating system at various levels, and the ability to perform autonomous tasks using browser automation. This integration involves configuring Ollama itself, managing necessary dependencies, setting up a persistent knowledge base from external sources, and defining a foundational system prompt for all hosted models. The core artifact produced is a PowerShell script designed to automate this entire setup process, granting Ollama models extensive control over the local machine environment.

1.2. Complexity and Security Considerations

The integration described herein represents a powerful but inherently complex undertaking. Granting an AI model the ability to execute arbitrary commands, access the entire filesystem, control web browsers, and interact with low-level system components introduces significant security risks. Malicious or unintended actions executed by the LLM could lead to data loss, system instability, unauthorized access, or exposure of sensitive information. Consequently, this configuration should only be deployed in controlled, isolated environments (such as dedicated virtual machines) for research and development purposes. Thorough understanding of the security implications and robust monitoring practices are paramount.[1] The principle of least privilege should be applied wherever possible, tailoring the script to disable capabilities not strictly required for the intended use case.

1.3. Report Structure

This document is structured to provide a comprehensive guide to the integration process:

- **Section 2:** Outlines prerequisites and configurable parameters within the automation script.
- **Section 3:** Details the core components and implementation steps, including privilege elevation, dependency management, Ollama setup, and the integration of various tools (web search, filesystem, command execution, browser automation, OS interaction).
- **Section 4:** Discusses the conceptual integration of the specified knowledge base.
- **Section 5:** Explores the advanced concept of reflection and how the script facilitates it.
- **Section 6:** Presents the complete PowerShell automation script with explanations.
- **Section 7:** Provides usage instructions, troubleshooting guidance, and final security recommendations.

2. Prerequisites and Configuration

2.1. System Requirements

Successful execution of the setup script and optimal performance of the Ollama agent depend on meeting specific hardware and software prerequisites:

- **Operating System:** Windows 10 (64-bit) or Windows 11.[4]
- **Processor:** Intel or AMD x86-64 CPU with SSE4.2 support. A recent multi-core processor (10th gen Intel or

equivalent AMD, or newer) is recommended for better performance.[4]

- **RAM:** Minimum 8GB, strongly recommended 16GB or more. For running larger models or multiple models concurrently, 32GB or higher is advised.[4]
- **Storage:** At least 10GB of free disk space for Ollama and base models. An SSD is recommended for faster model loading.[4] Additional space is required for cloned repositories and downloaded model files.
- **GPU (Optional but Recommended):** An NVIDIA GPU with CUDA support (e.g., RTX 3060 or better) significantly accelerates LLM inference.[4] Intel GPU support is experimental.[5]
- **PowerShell:** PowerShell version 5.1 (default on Windows 10/11) or preferably PowerShell 7+ for broader compatibility and features.[6]
- **Internet Connection:** Required for downloading Ollama, Git, dependencies, models, and cloning repositories.
- **Administrator Privileges:** The setup script requires execution with administrator rights to install software, modify system settings (like PATH), manage services, and potentially interact with protected system areas.[4]

2.2. Essential Dependencies

The automation script handles the installation or verification of the following key dependencies:

- **Git:** Required for cloning the specified GitHub repositories containing PowerShell functions and knowledge base materials.[9] The script will attempt to install Git if it's not detected.
- **Selenium PowerShell Module:** Used for browser automation tasks.[12] The script will install this module from the PowerShell Gallery.
- **Appropriate WebDriver:** Specifically, ChromeDriver is needed if Chrome browser automation is intended. The Selenium module often attempts to manage this, but manual installation or path specification might be required depending on the Chrome version.[15]

2.3. Script Configuration Parameters

The PowerShell automation script is designed with configurable parameters at the beginning, allowing users to customize the setup:

PowerShell

```
<#
.SYNOPSIS
Automates the setup of an Ollama instance on Windows with enhanced capabilities.

.DESCRIPTION
Installs Ollama, Git, and necessary PowerShell modules. Clones specified repositories
for functions and knowledge. Configures Ollama environment variables, sets a global
system prompt, pulls a default model, and integrates tools for web search, filesystem
access, command execution, browser automation, and OS interaction.

WARNING: This script grants significant system access to Ollama models.
Run only in isolated, controlled environments.
#>
param(
    # --- Repository Configuration ---
    [string]$PowerShellFunctionsRepo = "https://github.com/skyscope-sentinel/PowerShell.git",
```

```
    [string]$KnowledgeBaseRepo = "https://github.com/skyscope-sentinel/SKYSCOPESENTINELINTELLIGENCE-SELF-AWARE-AGI.git",
    [string]$LocalFunctionsPath = "$PSScriptRoot\PowerShellFunctions",
    [string]$LocalKnowledgePath = "$PSScriptRoot\KnowledgeBase",

    # --- Ollama Configuration ---
    [string]$OllamaInstallPath = "$env:ProgramFiles\Ollama", # Default install path [4]
    [string]$OllamaModelToPull = "llama3", # Default model to pull initially [17]
    [string]$OllamaHost = "127.0.0.1", # Host address Ollama should listen on [5]
    [string]$OllamaPort = "11434", # Port Ollama should listen on [5, 17]
    [string]$OllamaOrigins = "*", # Allowed origins for CORS (use '*' for any, or specific URLs) [5, 6, 7]
    [string]$OllamaKeepAlive = "15m", # How long models stay loaded in memory (e.g., "5m", "1h") [5]
    [string]$OllamaModelsDirectory = "$env:HOMEPATH\.ollama\models", # Default location, can be changed [5, 18, 19]
    [boolean]$SetOllamaDebug = $false, # Enable verbose debug logging for Ollama [5, 18]

    # --- Global System Prompt ---
    [string]$GlobalSystemPrompt = @"
You are a highly capable AI assistant integrated deeply into a Windows operating system via PowerShell.
You have been granted the following capabilities:
1. **Web Search:** You can search the internet for real-time information using the 'Invoke-WebSearch' function. Usage: Invoke-WebSearch -Query "search terms"
2. **Filesystem Access:** You can read, write, list, copy, move, and delete files and directories using standard PowerShell commands (e.g., Get-ChildItem, Get-Content, Set-Content, Copy-Item, Remove-Item, New-Item). Provide full paths. Example: Get-Content -Path "C:\Users\Admin\Documents\report.txt"
3. **Command Execution:** You can execute any PowerShell or CMD command using the 'Execute-SystemCommand' function. Specify the full command. Output and errors will be returned. Usage: Execute-SystemCommand -Command "Get-Process" OR Execute-SystemCommand -Command "cmd /c dir C:\"
4. **Browser Control (Chrome):** You can open URLs, navigate, and potentially interact with web pages using functions like 'Open-UrlInChrome'. Usage: Open-UrlInChrome -Url "https://github.com"
5. **OS Interaction:** You can manage Windows services (Start-Service, Stop-Service, Get-Service), query system information via WMI/CIM (Get-CimInstance), and interact with COM objects if needed via specific helper functions provided. Example: Execute-SystemCommand -Command "Get-Service -Name spooler"
6. **Knowledge Base:** You have access to a local knowledge base located at '$LocalKnowledgePath'. You can read files from this directory to answer questions based on its content. Example: Get-Content -Path "$LocalKnowledgePath\some_document.md"

Always prioritize using your integrated tools and local knowledge base when appropriate. Be precise and careful when executing commands or modifying files. State the actions you are taking clearly. If a request is ambiguous or potentially harmful, ask for clarification before proceeding.
"@,

    # --- Web Search Configuration ---
    [string]$SearchApiKey = "YOUR_SEARCH_API_KEY_HERE", # Replace with your actual SearchAPI.io key [20, 21]
    [string]$SearchApiEndpoint = "https://www.searchapi.io/api/v1/search",

    # --- Script Behavior ---
    [switch]$ForceReclone = $false, # If set, deletes existing local repos before cloning
    [switch]$SkipDependencyInstall = $false # If set, assumes Git and Selenium module are already installed
)

# --- (Rest of the script follows) ---
```

This parameter block allows tailoring the installation location, default model, network configuration (OLLAMA_HOST, OLLAMA_PORT, OLLAMA_ORIGINS), model persistence (OLLAMA_KEEP_ALIVE), model storage location (OLLAMA_MODELS), debug logging (SetOllamaDebug), the specific repositories to use, the content of the global system prompt, and the necessary API key for web search functionality.

3. Core Components and Implementation Strategy

This section details the crucial steps and PowerShell techniques employed by the automation script to achieve the desired Ollama integration.

3.1. Ensuring Elevated Privileges

Many actions performed by the script, such as installing software system-wide, modifying environment variables for all users, managing Windows services, and potentially accessing restricted filesystem locations, require administrator privileges.[4] Standard user accounts lack the necessary permissions for these operations.

To handle this, the script incorporates a check at the beginning to determine if it is running in an elevated context. It uses the System.Security.Principal.WindowsIdentity.NET class to get the current user's identity and checks if the user is in the built-in 'Administrators' group.[22]

PowerShell

```
# Function to check for Admin privileges
function Test-IsAdministrator {
    $currentUser = New-Object Security.Principal.WindowsPrincipal $(::GetCurrent())
    return $currentUser.IsInRole(::Administrator)
}

# Self-elevation logic
if (-not (Test-IsAdministrator)) {
    Write-Warning "Script requires administrator privileges. Attempting to relaunch elevated..."
    try {
        # Construct arguments, ensuring paths with spaces are quoted
        $scriptPath = "'$($MyInvocation.MyCommand.Definition)'" # Path to the current script
        $params = $MyInvocation.BoundParameters.GetEnumerator() | ForEach-Object { "-$($_.Key) '$($_.Value)'" }
        $allArgs = "-NoProfile -ExecutionPolicy Bypass -File $scriptPath $params"

        # Start new PowerShell process elevated
        Start-Process powershell.exe -Verb RunAs -ArgumentList $allArgs
        Write-Host "Elevated process launched."
        exit # Exit the current non-elevated process
    } catch {
        Write-Error "Failed to elevate privileges: $($_.Exception.Message)"
        exit 1
    }
} else {
    Write-Host "Running with Administrator privileges."
}
```

If the script detects it's not running as an administrator, it uses the Start-Process cmdlet with the -Verb RunAs parameter.[22] This action triggers the Windows User Account Control (UAC) prompt, requesting user confirmation to grant elevated permissions.[8] If the user approves, a new, elevated PowerShell instance is launched, executing the

same script with the original parameters. The non-elevated instance then exits. This ensures the rest of the script executes with the required permissions. Running processes with unnecessary elevation poses security risks; therefore, this elevated state should be used cautiously and only when necessary.[1]

3.2. Dependency Management: Git and Modules

The script relies on external resources and PowerShell modules. It automates their acquisition:

- **Git Installation:** It checks for the presence of git.exe in the system's PATH. If not found, it attempts to download and install Git for Windows silently using a package manager like Winget (if available) or by downloading the installer and running it. This ensures the git clone command is available.[26]
- **Repository Cloning:** Using the configured repository URLs ($PowerShellFunctionsRepo, $KnowledgeBaseRepo) and local paths ($LocalFunctionsPath, $LocalKnowledgePath), the script executes git clone.[9] The -RecurseSubmodules flag might be added if repositories contain submodules. The $ForceReclone switch allows overwriting existing local copies.

```
PowerShell
# Example Git Clone Logic
if ($ForceReclone -and (Test-Path $LocalFunctionsPath)) {
    Write-Host "Forcing reclone. Removing existing directory: $LocalFunctionsPath"
    Remove-Item -Path $LocalFunctionsPath -Recurse -Force
}
if (-not (Test-Path $LocalFunctionsPath)) {
    Write-Host "Cloning PowerShell Functions repository from $PowerShellFunctionsRepo..."
    git clone $PowerShellFunctionsRepo $LocalFunctionsPath
    if ($LASTEXITCODE -ne 0) { Write-Error "Git clone failed for Functions Repo."; exit 1 }
} else {
    Write-Host "Functions directory $LocalFunctionsPath already exists. Skipping clone unless -ForceReclone specified."
}
# Similar logic for KnowledgeBaseRepo...
```

- **PowerShell Module Installation:** The script checks if the Selenium module is installed using Get-Module -ListAvailable. If not found, it installs it from the PowerShell Gallery using Install-Module Selenium -Repository PSGallery -Force -SkipPublisherCheck -AllowClobber.[12] Error handling is included to manage potential installation issues.
- **Importing Local Functions:** Functions defined in .ps1 files within the cloned $LocalFunctionsPath directory need to be loaded into the current PowerShell session. The script uses dot-sourcing for this purpose. It iterates through .ps1 files in the specified directory and loads them.[27]

```
PowerShell
# Example Function Importing Logic
Write-Host "Importing functions from $LocalFunctionsPath..."
$functionFiles = Get-ChildItem -Path $LocalFunctionsPath -Filter *.ps1 -Recurse
foreach ($file in $functionFiles) {
    try {
        . $file.FullName
        Write-Verbose "Sourced function file: $($file.FullName)"
    } catch {
        Write-Warning "Failed to source function file '$($file.FullName)': $($_.Exception.Message)"
    }
}
```

Dot-sourcing executes the script in the current scope, making its functions available globally within the

session.[28] Using Import-Module against .ps1 files can lead to unexpected behavior on subsequent runs if not structured as proper modules (.psm1).[30]

3.3. Ollama Installation and Configuration

The script automates the setup and configuration of the Ollama service:

- **Installation:**
  1. Checks if ollama.exe exists in the target installation path ($OllamaInstallPath) or system PATH.
  2. If not found, downloads the official Ollama Windows installer (.exe) from ollama.ai.[4]
  3. Verifies the download (optional checksum validation).
  4. Executes the installer silently if possible, or guides the user through the interactive setup wizard. Running the installer with administrator privileges is crucial.[4] The installer typically sets up Ollama as a background service configured to start on boot.[6]
- **PATH Configuration:** Ensures the Ollama installation directory is added to the system or user PATH environment variable. This allows running ollama commands from any terminal location.[4] This is often handled by the installer but can be done manually via PowerShell if needed:

```
PowerShell
# Example PATH update (System-wide, requires elevation)
$currentPath = [Environment]::GetEnvironmentVariable('Path', 'Machine')
if ($currentPath -notlike "*$OllamaInstallPath*") {
    Write-Host "Adding Ollama to System PATH..."
    $newPath = "$currentPath;$OllamaInstallPath"
    [Environment]::SetEnvironmentVariable('Path', $newPath, 'Machine')
    Write-Host "Ollama added to PATH. A system restart or new session might be needed for changes to take effect."
}
```

- **Verification:** Runs ollama --version to confirm successful installation.[4] Uses curl http://localhost:11434 (or the configured host/port) to check if the Ollama service is running.[6]
- **Model Pulling:** Executes ollama pull $OllamaModelToPull to download the specified default model (e.g., llama3).[4]
- **Environment Variable Configuration:** Sets essential Ollama environment variables using the setx command or by directly modifying registry keys for persistence across sessions. This requires administrator privileges for system-wide settings (/m flag with setx).[6]
  - OLLAMA_HOST: Configures the listening IP address (e.g., 127.0.0.1 for local only, 0.0.0.0 for all interfaces) and port.[5] The script combines $OllamaHost and $OllamaPort parameters.
  - OLLAMA_MODELS: Specifies the directory where Ollama stores model files. Allows moving models off the system drive.[5] Set using the $OllamaModelsDirectory parameter.
  - OLLAMA_ORIGINS: Defines allowed origins for Cross-Origin Resource Sharing (CORS). Crucial for web-based applications interacting with Ollama.[5] Set using the $OllamaOrigins parameter (e.g., * or http://localhost:8080,https://mywebapp.com).
  - OLLAMA_KEEP_ALIVE: Controls how long models remain loaded in memory after inactivity. Longer times improve responsiveness for frequent requests but consume more RAM.[5] Set using the $OllamaKeepAlive parameter.
  - OLLAMA_DEBUG: Enables verbose logging ($true/1 or $false/0) for troubleshooting.[5] Set based on the

$SetOllamaDebug switch.

```powershell
# Example Environment Variable Setting (User scope)
Write-Host "Setting Ollama environment variables..."
setx OLLAMA_HOST "$($OllamaHost):$($OllamaPort)"
setx OLLAMA_ORIGINS "$OllamaOrigins"
setx OLLAMA_KEEP_ALIVE "$OllamaKeepAlive"
if ($OllamaModelsDirectory -ne "$env:HOMEPATH\.ollama\models") { # Only set if different from default
    if (-not (Test-Path $OllamaModelsDirectory)) { New-Item -Path $OllamaModelsDirectory -ItemType Directory -Force | Out-Null }
    setx OLLAMA_MODELS "$OllamaModelsDirectory"
}
setx OLLAMA_DEBUG (if ($SetOllamaDebug) { "1" } else { "0" })

Write-Host "Environment variables set. Restarting Ollama service to apply changes..."
# Restart logic depends on how Ollama runs (Service, background process)
# Example for service: Restart-Service -Name OllamaService (if named OllamaService)
# Or stop/start the Ollama background process if run manually/via app
```

Restarting the Ollama service or application is necessary after setting environment variables for them to take effect.[31]

- **Global System Prompt Configuration:** Setting a system prompt via environment variables (OLLAMA_SYSTEM_PROMPT) might not be directly supported or might be overridden. A more reliable method for setting a *default* system prompt for a specific model is to modify its Modelfile.
    1. Create a custom Modelfile (e.g., Modelfile_CustomLlama3).
    2. Use the FROM instruction to base it on the desired model (FROM $OllamaModelToPull).
    3. Add the SYSTEM instruction with the content of the $GlobalSystemPrompt parameter.[33]
    4. Use ollama create custom-$OllamaModelToPull -f Modelfile_CustomLlama3 to build a new model variant with the embedded system prompt.
    5. Instruct the user (or potentially configure the script's helper functions) to use this new custom-$OllamaModelToPull model name for interactions.

```powershell
# Example Modelfile creation and model building
Write-Host "Creating custom model with global system prompt..."
$modelfileContent = @"
FROM $OllamaModelToPull
SYSTEM """$GlobalSystemPrompt"""
"@ # Using triple quotes handles multi-line prompts

$modelfilePath = Join-Path $PSScriptRoot "Modelfile_Custom_$($OllamaModelToPull)"
$modelfileContent | Out-File -FilePath $modelfilePath -Encoding UTF8

$customModelName = "custom-$OllamaModelToPull"
ollama create $customModelName -f $modelfilePath

if ($LASTEXITCODE -eq 0) {
    Write-Host "Successfully created custom model '$customModelName' with the specified system prompt."
    Write-Host "Use this model name ('$customModelName') for interactions."
} else {
    Write-Warning "Failed to create custom model '$customModelName'. The global system prompt might not be applied by default."
}
# Clean up Modelfile
Remove-Item -Path $modelfilePath -Force
```

This approach ensures the system prompt is consistently applied when using the custom model variant, addressing potential limitations with environment variables or per-request prompt settings.[34]

3.4. Tool Integration Strategies

The script integrates various tools by defining PowerShell functions that the LLM can invoke via the

Execute-SystemCommand mechanism or potentially through more structured function calling if Ollama's API supports it directly in the future.

- **3.4.1. Web Search:**
  - **Mechanism:** Implements a PowerShell function (e.g., Invoke-WebSearch) that takes a query string.
  - **Implementation:** This function constructs a request to an external search API (like SearchAPI.io mentioned in [20]). It uses Invoke-RestMethod or Invoke-WebRequest to send the query and the API key ($SearchApiKey) to the $SearchApiEndpoint.
  - **Output:** Parses the JSON response from the API, extracts relevant results (snippets, URLs, titles), and returns them as a formatted string or PowerShell object for the LLM. Error handling for API failures or invalid keys is included.
  - **LLM Interaction:** The global system prompt explicitly tells the LLM how to use this function (Invoke-WebSearch -Query "..."). The Execute-SystemCommand function acts as the bridge.

```
PowerShell
# Example Web Search Function (Simplified)
function Invoke-WebSearch {
    param(
        [Parameter(Mandatory=$true)]
        [string]$Query
    )
    if ($SearchApiKey -eq "YOUR_SEARCH_API_KEY_HERE" -or -not $SearchApiKey) {
        return "Error: Web search requires a valid Search API Key configured in the script parameters."
    }
    $headers = @{ "Authorization" = "Bearer $SearchApiKey" } # Adjust based on API spec
    $uri = "$SearchApiEndpoint`?q=$([uri]::EscapeDataString($Query))&engine=google" # Example

    try {
        $response = Invoke-RestMethod -Uri $uri -Method Get #-Headers $headers # Headers might not be needed depending on API
        # Process $response (which is likely JSON) to extract and format results
        # Example: return $response.organic_results | Select-Object -First 5 | Format-Table | Out-String
         return $response.organic_results.snippet -join "`n`n" # Return top snippets as string
    } catch {
        return "Error during web search: $($_.Exception.Message)"
    }
}
```

- **3.4.2. Filesystem Access:**
  - **Mechanism:** Leverages built-in PowerShell cmdlets.
  - **Implementation:** The LLM can request execution of standard filesystem commands via Execute-SystemCommand. Examples include:
    - Get-ChildItem -Path C:\Users\Admin\Downloads (List files) [35]
    - Get-Content -Path C:\data\config.txt (Read file) [35]
    - Set-Content -Path C:\output\log.txt -Value "Log entry" (Write file) [36]
    - Copy-Item -Path C:\source\file.zip -Destination D:\backup\ (Copy) [35]
    - Remove-Item -Path C:\temp\oldfile.txt -Force (Delete) [35]
    - New-Item -Path C:\temp\NewFolder -ItemType Directory (Create folder) [35]
    - Test-Path -Path \\server\share\file.dat (Check existence, supports UNC paths) [35]
  - **LLM Interaction:** The system prompt guides the LLM on how to use these commands within the Execute-SystemCommand function. The script ensures the output (or errors) from these commands is captured and returned.
- **3.4.3. Command Execution:**

- **Mechanism:** A core function, Execute-SystemCommand, is defined to run arbitrary commands.
- **Implementation:**
  - Uses Invoke-Expression for executing PowerShell commands directly within the current runspace. This is powerful but carries risks if the command string is not carefully controlled.
  - Alternatively, uses Start-Process with -NoNewWindow, -Wait, -RedirectStandardOutput, -RedirectStandardError to run external executables (.exe) or cmd.exe /c for batch commands, capturing their output streams reliably.[38] This is generally safer than Invoke-Expression for arbitrary strings.
  - Captures both standard output and standard error streams and returns them to the LLM.[38]
- **LLM Interaction:** The system prompt explicitly defines how to use Execute-SystemCommand. The LLM formulates the command string (e.g., Get-Process, ipconfig, python script.py).

```
PowerShell
# Example Command Execution Function (using Start-Process for safety)
function Execute-SystemCommand {
    param(
        [Parameter(Mandatory=$true)]
        [string]$Command
    )
    Write-Verbose "Executing command: $Command"
    $tempOutFile = New-TemporaryFile
    $tempErrFile = New-TemporaryFile

    try {
        # Determine if it's a PowerShell command or external executable/cmd
        # Simple heuristic: check if command exists as PS command, else treat as external
        $isPsCommand = Get-Command $Command.Split(' ') -ErrorAction SilentlyContinue

        if ($isPsCommand) {
            # Execute PowerShell command using Invoke-Command for isolation (safer than Invoke-Expression)
            # Or potentially just Invoke-Expression if full runspace access is intended (use with caution)
            $process = Invoke-Command -ScriptBlock { Invoke-Expression $using:Command } *>&1 | Out-File -FilePath
$tempOutFile.FullName -Encoding utf8 -Force
            # Note: Capturing error stream separately for Invoke-Expression is tricky. This merges streams.
            $stdOut = Get-Content $tempOutFile.FullName -Raw
            $stdErr = "" # Error merged into stdout with *>&1
            <span class="math-inline">exitCode \= if \(</span>?) { 0 } else { $LASTEXITCODE | 1 } # Best guess for exit code
        } else {
            # Assume external command, use cmd /c for shell features or direct execution
            $processInfo = New-Object System.Diagnostics.ProcessStartInfo
            $processInfo.FileName = "cmd.exe"
            $processInfo.Arguments = "/c $Command" # Use cmd /c to handle shell commands like dir, echo etc.
            $processInfo.RedirectStandardOutput = $true
            $processInfo.RedirectStandardError = $true
            $processInfo.UseShellExecute = $false
            $processInfo.CreateNoWindow = $true

            $process = New-Object System.Diagnostics.Process
            $process.StartInfo = $processInfo
            $process.Start() | Out-Null

            $stdOut = $process.StandardOutput.ReadToEnd()
            $stdErr = $process.StandardError.ReadToEnd()

            $process.WaitForExit()
            $exitCode = $process.ExitCode

            # Write streams to temp files for consistency if needed, or just use variables
```

```
            $stdOut | Out-File -FilePath $tempOutFile.FullName -Encoding utf8 -Force
            $stdErr | Out-File -FilePath $tempErrFile.FullName -Encoding utf8 -Force
        }

        $output = "Exit Code: $exitCode`n"
        if ($stdOut) { $output += "Standard Output:`n$stdOut`n" }
        if ($stdErr) { $output += "Standard Error:`n$stdErr`n" }

        return $output.Trim()

    } catch {
        return "Error executing command '$Command': $($_.Exception.Message)"
    } finally {
        Remove-Item $tempOutFile.FullName -ErrorAction SilentlyContinue
        Remove-Item $tempErrFile.FullName -ErrorAction SilentlyContinue
    }
}
```

This function provides a controlled way for the LLM to interact with the command line, capturing results essential for feedback and subsequent actions. The choice between Invoke-Expression and Start-Process involves a trade-off between direct PowerShell integration and security/isolation.

- **3.4.4. Browser Automation:**
  - **Mechanism:** Primarily uses the Selenium PowerShell module.[13] This module provides cmdlets to control web browsers like Chrome.
  - **Implementation:** Wrapper functions like Open-UrlInChrome, Get-WebPageContent, or Click-WebElement are created. These functions internally use Selenium cmdlets:
    - Start-SeChrome: Launches and controls Chrome (potentially headless or with specific profiles).[13] Requires matching ChromeDriver.[15]
    - Enter-SeUrl: Navigates to a specific URL.[13]
    - Find-SeElement: Locates elements using selectors (ID, Name, CSS, XPath).[12]
    - Invoke-SeClick: Clicks elements.[13]
    - Send-SeKeys: Enters text into fields.[12]
    - Get-SeElement: Retrieves element properties or text.
    - Stop-SeDriver: Closes the browser session.
  - **Alternative (Advanced):** Direct interaction with the Chrome DevTools Protocol (CDP).[41] While powerful, there isn't a mature, dedicated PowerShell client library readily available.[41] Implementing CDP interaction would likely involve complex WebSocket communication or leveraging.NET libraries like Microsoft.Web.WebView2.DevToolsProtocolExtension [43] if running within a WebView2 context, or potentially bridging to Puppeteer (Node.js) or Playwright (Python/.NET).[41] The Selenium module provides a more accessible PowerShell-native approach.
  - **LLM Interaction:** The system prompt describes the available browser functions (e.g., Open-UrlInChrome -Url "..."). The LLM requests these actions via Execute-SystemCommand.
- **3.4.5. Operating System Interaction:**
  - **Mechanism:** Leverages various built-in PowerShell capabilities for deeper OS control.
  - **Implementation:** Exposes specific OS interactions through wrapper functions or allows direct command execution via Execute-SystemCommand.
    - **Windows Services:** Use Get-Service, Start-Service, Stop-Service, Restart-Service, Set-Service cmdlets.[44] Example: Execute-SystemCommand -Command "Restart-Service -Name Spooler -Force"
    - **WMI/CIM:** Use Get-CimInstance to query system information (hardware, software, OS configuration).[47]

Example: Execute-SystemCommand -Command "Get-CimInstance -ClassName Win32_Processor"

- **COM Objects:** Use New-Object -ComObject to interact with COM components like WScript.Shell (for shortcuts, popups) or application-specific objects (e.g., Excel.Application).[50] Wrapper functions should be created for specific, safe COM tasks. Example: A function Create-DesktopShortcut -TargetPath "C:\MyApp.exe" -ShortcutName "My App" could use WScript.Shell.
- **P/Invoke (Advanced):** For low-level access, define C# signatures for Windows API functions and use Add-Type or reflection techniques (like the PSReflect module concept) to call them.[52] This is complex and should be reserved for specific needs not covered by higher-level cmdlets. The script could pre-define P/Invoke wrappers for specific, safe API calls if required by the user query.

- **LLM Interaction:** The system prompt lists available OS interaction functions or guides the LLM to use standard cmdlets via Execute-SystemCommand. The level of access can be graduated by selectively implementing and exposing wrappers for these different mechanisms. This tiered approach provides the requested depth while managing complexity.

3.5. Global System Prompt Integration (Revisited)

As detailed in Section 3.3, the most robust way to ensure the detailed system prompt ($GlobalSystemPrompt) is consistently used is by embedding it into a custom model variant using ollama create. The prompt itself (defined in Section 2.3) is crucial as it informs the LLM about its identity, capabilities (available tools/functions and how to use them), limitations, and expected behavior within the integrated Windows environment. It explicitly lists the functions like Invoke-WebSearch, Execute-SystemCommand, Open-UrlInChrome, and guides the use of standard PowerShell cmdlets for filesystem and OS tasks.

4. Knowledge Stack Integration

The user query requests the integration of a "knowledge stack" from the materials contained in the SKYSCOPESENTINELINTELLIGENCE-SELF-AWARE-AGI repository [user query]. The PowerShell script facilitates the *availability* of this knowledge but does not implement the full Retrieval-Augmented Generation (RAG) pipeline typically associated with using such a knowledge base effectively.

4.1. Making Knowledge Available

The script achieves the first step by cloning the specified knowledge base repository ($KnowledgeBaseRepo) to a local directory ($LocalKnowledgePath).[9] This ensures all the text files, documents, markdown files, code, etc., within that repository are present on the local filesystem.

4.2. How Ollama Can Use the Knowledge

Once the files are local, the Ollama agent can access them using the integrated filesystem tools:

1. **Direct File Reading:** The LLM can be instructed (via the system prompt or user request) to read specific files from the $LocalKnowledgePath using Get-Content executed via Execute-SystemCommand. Example prompt: "Read the document $LocalKnowledgePath\core_principles.md and summarize its main points."
2. **Listing Contents:** The LLM can explore the knowledge base structure using Get-ChildItem -Path $LocalKnowledgePath to understand available files and directories.

4.3. Limitations and RAG Considerations

This approach has significant limitations compared to a true RAG system:

- **Scalability:** Reading entire files into the LLM's context window for every query is inefficient and often

impossible due to context length limits, especially for large documents or knowledge bases.[55]

- **Relevance:** The LLM doesn't automatically know *which* file or section contains the relevant information for a given query. It relies on explicit instructions or potentially inefficient browsing.
- **No Semantic Search:** Simple file reading lacks semantic understanding. RAG systems typically use embedding models to create vector representations of knowledge chunks, allowing for efficient searching based on meaning rather than just keywords.[55]

True RAG Implementation: Implementing RAG would involve:

1. **Chunking:** Breaking down the documents in $LocalKnowledgePath into smaller, manageable chunks.
2. **Embedding:** Using an embedding model (potentially another Ollama model or a dedicated one) to generate vector embeddings for each chunk.
3. **Indexing:** Storing these embeddings in a vector database (e.g., ChromaDB, FAISS).
4. **Retrieval:** When a query comes in, embedding the query and searching the vector database for the most semantically similar chunks.
5. **Augmentation:** Providing these retrieved chunks as context to the primary LLM along with the original query.

This RAG process is typically orchestrated using frameworks like LangChain or LlamaIndex, often in Python.[55] While the PowerShell script makes the data *available*, building the RAG pipeline itself is outside its scope but is a logical next step for leveraging the cloned knowledge base effectively. The system prompt informs the LLM of the *location* of the knowledge base, enabling basic file access as a starting point.

5. Exploring Advanced Concepts: Embedding Reflection and Self-Reflection

The user query mentions implementing "embedding reflection and other possible implementations". This likely refers to advanced agentic techniques aimed at improving LLM reasoning and reliability.

5.1. Conceptual Overview

- **Embedding Reflection (Less Common/Specific):** This term isn't standard in mainstream LLM agent literature. It *could* refer to:
  - Techniques analyzing semantic relationships within embedding spaces, potentially to improve negative sampling in contrastive learning for better sentence representations.[56]
  - Models manipulating or reasoning about their own internal vector representations (embeddings).
  - Using embeddings for more nuanced tasks beyond simple similarity search. Without further clarification, this interpretation remains speculative.
- **Self-Reflection (Common Agentic Technique):** This refers to frameworks where an AI agent critiques its own outputs, reasoning processes, or action plans to identify flaws and iteratively improve performance.[59] It mimics human "System 2" thinking – pausing to analyze and correct initial "System 1" responses.[61] Key frameworks include:
  - **Reflexion:** Uses linguistic feedback (self-generated critique) as a form of "verbal reinforcement" to guide the agent's next attempt. It involves an Actor (generates action/text), an Evaluator (scores the output), and a Self-Reflection model (generates critique based on trajectory, reward, and memory).[60] It often builds upon frameworks like ReAct.
  - **ReAct (Reason+Act):** Interleaves reasoning steps (thought process) with actions (using tools). While not

strictly reflection, the reasoning steps provide a basis for potential later reflection.[60]

- **Language Agent Tree Search (LATS):** Combines reflection/evaluation with Monte Carlo Tree Search to explore different action trajectories, using feedback to guide the search towards better solutions.[58]
- **Simple Reflection Loops:** Involve prompting an LLM to generate a response, then prompting the same or another LLM (perhaps with a "critic" persona) to critique it, feeding the critique back for refinement.[61]

Self-Reflection is the more established and actionable concept in the context of building capable LLM agents.

5.2. How the PowerShell Script Facilitates Reflection

The PowerShell script presented here does **not** implement the multi-step reflection loop logic itself. These loops typically involve multiple LLM calls and state management, often orchestrated by frameworks like LangChain (using LangGraph [61]) or LlamaIndex [57], usually in Python.

However, the script provides the essential **foundational capabilities** that enable such reflection processes:

- **Tools for Grounding:** Reflection often requires grounding the critique in facts or verifying steps. The integrated tools – web search (Invoke-WebSearch), command execution (Execute-SystemCommand), filesystem access, and browser control – provide the means for the agent to gather external information during its reflection phase.[61] For example, a reflecting agent could use Invoke-WebSearch to check a fact it stated or Execute-SystemCommand to verify the outcome of a command it previously ran.
- **System Prompt Encouragement:** The $GlobalSystemPrompt can be tailored to explicitly encourage reflective behavior. Phrases like "Review your steps before finalizing", "Verify facts using web search if unsure", or "Consider alternative approaches" can nudge the LLM towards more careful, System-2-like processing.
- **Action Execution:** The ability to execute actions (commands, file operations, etc.) is fundamental for any agent, including reflective ones, to interact with the environment based on their plans or refined strategies.
- **Implicit Memory:** While lacking sophisticated memory modules, the filesystem access allows a rudimentary form of memory persistence. A highly customized reflection loop could potentially direct the agent to write its thoughts, critiques, or past trajectories to files in a designated directory, which could then be read back in subsequent steps. This is less efficient than in-memory databases or dedicated memory buffers used by frameworks like LlamaIndex [57] but offers a basic mechanism enabled by the script's tools.

Therefore, the PowerShell script acts as a crucial **enabler** for advanced agentic techniques like self-reflection. It sets up the environment and provides the necessary tools and access points that more complex orchestration logic (likely external to this script) can leverage to implement reflection loops, allowing the Ollama agent to perform tasks more reliably and accurately.

6. The Complete PowerShell Automation Script

(Note: The following script is a comprehensive representation based on the preceding analysis. It includes placeholders for API keys and requires careful review and adaptation before execution. Error handling is included but may need further refinement for production environments. It assumes the user will run it in an elevated PowerShell session after configuring the parameters.)

PowerShell

```
<#
.SYNOPSIS
Automates the setup of an Ollama instance on Windows with enhanced capabilities including web search, filesystem access, command execution,
browser automation, and deep OS integration.

.DESCRIPTION
Installs Ollama, Git, and necessary PowerShell modules (Selenium). Clones specified repositories
for PowerShell functions and knowledge base materials. Configures Ollama environment variables,
sets a global system prompt by creating a custom model variant, pulls a default model,
and defines wrapper functions for integrated tools.

WARNING: This script grants significant system access and control to Ollama models.
It installs software and modifies system settings (PATH, Environment Variables).
Execute ONLY in a secure, isolated environment (e.g., dedicated VM) for development/testing.
Review the script carefully before running. You are responsible for its use.

Requires running as Administrator. It will attempt to self-elevate if not already admin.
#>

param(
    # --- Repository Configuration ---
    [string]$PowerShellFunctionsRepo = "https://github.com/skyscope-sentinel/PowerShell.git",
    [string]$KnowledgeBaseRepo = "https://github.com/skyscope-sentinel/SKYSCOPESENTINELINTELLIGENCE-SELF-AWARE-AGI.git",
    [string]$LocalFunctionsPath = "$PSScriptRoot\PowerShellFunctions",
    [string]$LocalKnowledgePath = "$PSScriptRoot\KnowledgeBase",

    # --- Ollama Configuration ---
    [string]$OllamaInstallDir = "$env:ProgramFiles\Ollama", # Default install directory [4]
    [string]$OllamaModelToPull = "llama3", # Default model to pull initially [6, 17]
    [string]$OllamaHost = "127.0.0.1", # Host address Ollama should listen on [5]
    [string]$OllamaPort = "11434", # Port Ollama should listen on [5, 17]
    [string]$OllamaOrigins = "*", # Allowed origins for CORS ('*' or comma-separated URLs) [5, 6, 7]
    [string]$OllamaKeepAlive = "15m", # How long models stay loaded (e.g., "5m", "1h") [5]
    [string]$OllamaModelsDirectory = "$env:USERPROFILE\.ollama", # Default base dir for models [5, 18, 19] - Note: Ollama stores models
under.\models within this path.
    [boolean]$SetOllamaDebug = $false, # Enable verbose debug logging for Ollama [5, 18]

    # --- Global System Prompt ---
    # Note: $LocalKnowledgePath variable will be expanded before embedding.
    [string]$GlobalSystemPromptTemplate = @"
You are a highly capable AI assistant integrated deeply into a Windows operating system via PowerShell.
You have been granted the following capabilities:
1.  **Web Search:** You can search the internet for real-time information using the 'Invoke-WebSearch' function. Usage: Invoke-WebSearch -Query
"search terms"
2.  **Filesystem Access:** You can read, write, list, copy, move, and delete files and directories using standard PowerShell commands (e.g.,
Get-ChildItem, Get-Content, Set-Content, Copy-Item, Remove-Item, New-Item). Provide full paths. Example: Get-Content -Path
"C:\Users\Admin\Documents\report.txt"
3.  **Command Execution:** You can execute any PowerShell or CMD command using the 'Execute-SystemCommand' function. Specify the full
command. Output and errors will be returned. Usage: Execute-SystemCommand -Command "Get-Process" OR Execute-SystemCommand
-Command "cmd /c dir C:\"
4.  **Browser Control (Chrome):** You can open URLs in Chrome using 'Open-UrlInChrome'. More advanced browser interaction might require
specific Selenium commands via Execute-SystemCommand if wrappers aren't available. Usage: Open-UrlInChrome -Url "https://github.com"
5.  **OS Interaction:** You can manage Windows services (Start-Service, Stop-Service, Get-Service), query system information via WMI/CIM
(Get-CimInstance), and interact with COM objects if needed via specific helper functions provided or standard PowerShell cmdlets. Example:
Execute-SystemCommand -Command "Get-Service -Name spooler"
6.  **Knowledge Base:** You have access to a local knowledge base located at '{0}'. You can read files from this directory to answer questions
based on its content. Example: Get-Content -Path "{0}\some_document.md"

Always prioritize using your integrated tools and local knowledge base when appropriate. Be precise and careful when executing commands or
modifying files. State the actions you are taking clearly. If a request is ambiguous or potentially harmful, ask for clarification before proceeding.
```

```
"@,

    # --- Web Search Configuration ---
    [string]$SearchApiKey = "YOUR_SEARCH_API_KEY_HERE", # *** REPLACE WITH YOUR ACTUAL SearchAPI.io KEY *** [20, 21]
    [string]$SearchApiEndpoint = "https://www.searchapi.io/api/v1/search", # Example endpoint [20]

    # --- Script Behavior ---
    [switch]$ForceReclone = $false, # If set, deletes existing local repos before cloning
    [switch]$SkipDependencyInstall = $false, # If set, assumes Git and Selenium module are already installed
    [switch]$SkipOllamaInstall = $false, # If set, assumes Ollama is already installed
    [switch]$SkipCustomModelCreation = $false # If set, skips creating the model variant with the embedded prompt
)

# ================================================================================
# PRE-FLIGHT CHECKS AND SETUP
# ================================================================================
Write-Host "Starting Ollama Deep Integration Setup Script..." -ForegroundColor Yellow

# Check for Administrator Privileges and Self-Elevate if Necessary
function Test-IsAdministrator {
    $currentUser = New-Object Security.Principal.WindowsPrincipal $(::GetCurrent())
    return $currentUser.IsInRole(::Administrator)
}

if (-not (Test-IsAdministrator)) {
    Write-Warning "Script requires administrator privileges. Attempting to relaunch elevated..."
    try {
        $scriptPath = "'$($MyInvocation.MyCommand.Definition)'"
        # Rebuild arguments carefully, quoting values
        $paramArgs = $PSBoundParameters.Keys | ForEach-Object {
            $key = $_
            $value = $PSBoundParameters[$key]
            if ($value -is [switch]) { # Handle switches
                if ($value.IsPresent) { "-$key" }
            } elseif ($value -is [string] -or $value -is [boolean]) {
                "-$key '$value'" # Quote strings and booleans
            } else {
                "-$key $value" # Pass others directly (e.g., numbers)
            }
        }
        $allArgs = "-NoProfile -ExecutionPolicy Bypass -File $scriptPath $($paramArgs -join ' ')"

        Start-Process powershell.exe -Verb RunAs -ArgumentList $allArgs
        Write-Host "Elevated process launched. Exiting current process."
        exit 0 # Exit current non-elevated process successfully
    } catch {
        Write-Error "Failed to elevate privileges: $($_.Exception.Message)"
        Write-Error "Please run this script manually from an Administrator PowerShell session."
        exit 1
    }
} else {
```

```
    Write-Host "Running with Administrator privileges."
}

# Set Error Action Preference
$ErrorActionPreference = 'Stop'

# Ensure script root directory exists for relative paths
if (-not (Test-Path $PSScriptRoot)) {
    Write-Warning "PSScriptRoot not defined. Setting to current directory."
    $PSScriptRoot = Get-Location
}
$LocalFunctionsPath = Join-Path $PSScriptRoot "PowerShellFunctions"
$LocalKnowledgePath = Join-Path $PSScriptRoot "KnowledgeBase"

# ============================================================================
# DEPENDENCY INSTALLATION
# ============================================================================
Write-Host "`n=== Installing Dependencies ===" -ForegroundColor Cyan

if (-not $SkipDependencyInstall) {
    # Check/Install Git
    Write-Host "Checking for Git..."
    $gitPath = Get-Command git -ErrorAction SilentlyContinue
    if (-not $gitPath) {
        Write-Warning "Git not found. Attempting to install using Winget..."
        try {
            winget install --id Git.Git -e --source winget --accept-package-agreements --accept-source-agreements
            $gitPath = Get-Command git -ErrorAction SilentlyContinue
            if (-not $gitPath) { throw "Git installation via Winget failed or Git is not in PATH." }
            Write-Host "Git installed successfully via Winget." -ForegroundColor Green
        } catch {
            Write-Error "Failed to install Git automatically: $($_.Exception.Message)"
            Write-Error "Please install Git manually from https://git-scm.com/download/win and ensure it's in your PATH."
            exit 1
        }
    } else {
        Write-Host "Git found at $($gitPath.Source)"
    }

    # Check/Install Selenium Module
    Write-Host "Checking for Selenium PowerShell Module..."
    if (-not (Get-Module -ListAvailable -Name Selenium)) {
        Write-Warning "Selenium module not found. Installing from PSGallery..."
        try {
            Install-Module Selenium -Repository PSGallery -Force -SkipPublisherCheck -AllowClobber -Scope CurrentUser # Install for
current user to avoid potential system-wide issues
            Write-Host "Selenium module installed successfully." -ForegroundColor Green
        } catch {
            Write-Error "Failed to install Selenium module: $($_.Exception.Message)"
            Write-Error "Please ensure PowerShellGet is updated (`Install-Module PowerShellGet -Force`) and try again."
            # Consider adding PSGallery registration steps if needed
            exit 1
```

```
        }
    } else {
        Write-Host "Selenium module found."
    }
    # Import Selenium module for potential use later in script (e.g., driver setup hints)
    Import-Module Selenium -ErrorAction SilentlyContinue

} else {
    Write-Host "Skipping dependency installation as requested."
}

# ================================================================================
# REPOSITORY CLONING
# ================================================================================
Write-Host "`n=== Cloning Repositories ===" -ForegroundColor Cyan

function Clone-Repository ([string]$RepoUrl, [string]$LocalPath, [switch]$Force) {
    if ($Force -and (Test-Path $LocalPath)) {
        Write-Host "Force reclone enabled. Removing existing directory: $LocalPath"
        try {
            Remove-Item -Path $LocalPath -Recurse -Force
        } catch {
            Write-Warning "Could not remove existing directory '$LocalPath': $($_.Exception.Message)"
        }
    }
    if (-not (Test-Path $LocalPath)) {
        Write-Host "Cloning repository from $RepoUrl to $LocalPath..."
        try {
            git clone $RepoUrl $LocalPath # Add --depth 1 for shallow clone if full history isn't needed
            if ($LASTEXITCODE -ne 0) { throw "Git clone command failed with exit code $LASTEXITCODE" }
            Write-Host "Repository cloned successfully." -ForegroundColor Green
        } catch {
            Write-Error "Git clone failed for '$RepoUrl': $($_.Exception.Message)"
            # Consider adding specific error handling for authentication issues if using private repos
            exit 1
        }
    } else {
        Write-Host "Directory $LocalPath already exists. Skipping clone. Use -ForceReclone to overwrite."
    }
}

Clone-Repository -RepoUrl $PowerShellFunctionsRepo -LocalPath $LocalFunctionsPath -Force:$ForceReclone
Clone-Repository -RepoUrl $KnowledgeBaseRepo -LocalPath $LocalKnowledgePath -Force:$ForceReclone

# ================================================================================
# OLLAMA INSTALLATION & CONFIGURATION
# ================================================================================
Write-Host "`n=== Setting up Ollama ===" -ForegroundColor Cyan

# --- Installation ---
if (-not $SkipOllamaInstall) {
    Write-Host "Checking for existing Ollama installation..."
```

```powershell
    $ollamaExePath = Join-Path $OllamaInstallDir "ollama.exe"
    $ollamaInPath = Get-Command ollama -ErrorAction SilentlyContinue

    if (-not (Test-Path $ollamaExePath) -and -not $ollamaInPath) {
        Write-Warning "Ollama not found. Attempting to download and install..."
        $installerUrl = "https://ollama.com/download/OllamaSetup.exe" # Verify this URL is current
        $installerPath = Join-Path $env:TEMP "OllamaSetup.exe"
        try {
            Write-Host "Downloading Ollama installer from $installerUrl..."
            Invoke-WebRequest -Uri $installerUrl -OutFile $installerPath
            Write-Host "Download complete. Running installer (requires admin rights)..."
            # Attempt silent install if possible, otherwise user interaction needed
            Start-Process -FilePath $installerPath -ArgumentList "/S" -Wait -Verb RunAs # /S might be silent flag, check installer docs
            # Add verification step here - check if $ollamaExePath now exists
             if (-not (Test-Path $ollamaExePath)) {
                Start-Sleep -Seconds 5 # Give installer time
                if (-not (Test-Path $ollamaExePath)) {
                    Write-Warning "Silent install might have failed or requires interaction. Please check."
                    # Fallback to interactive install if silent fails? Or just error out?
                    # Start-Process -FilePath $installerPath -Wait -Verb RunAs
                }
             }
            Write-Host "Ollama installation process initiated." -ForegroundColor Green
            # Installer likely adds to PATH and starts service, but we verify/configure below
        } catch {
            Write-Error "Failed to download or install Ollama: $($_.Exception.Message)"
            Write-Error "Please install Ollama manually from https://ollama.com/"
            exit 1
        } finally {
            if (Test-Path $installerPath) { Remove-Item $installerPath -Force }
        }
    } else {
        Write-Host "Ollama installation detected."
    }
} else {
    Write-Host "Skipping Ollama installation check as requested."
}

# --- PATH Verification/Configuration (Installer usually handles this) ---
Write-Host "Verifying Ollama PATH..."
$envPaths = $env:PATH -split ';'
if ($envPaths -notcontains $OllamaInstallDir) {
    Write-Warning "Ollama directory '$OllamaInstallDir' not found in user PATH. Adding it..."
    # Add to User PATH for persistence without requiring immediate reboot
    $currentUserPath = [Environment]::GetEnvironmentVariable('Path', 'User')
    if ($currentUserPath -notlike "*$OllamaInstallDir*") {
        $newUserPath = "$currentUserPath;$OllamaInstallDir"
        [Environment]::SetEnvironmentVariable('Path', $newUserPath, 'User')
        Write-Host "Added to User PATH. You may need to restart PowerShell/Terminal for changes to apply."
        # Update current session's PATH
        $env:PATH += ";$OllamaInstallDir"
```

```
    } else {
        Write-Host "Ollama directory already in User PATH."
    }
} else {
    Write-Host "Ollama directory found in PATH."
}


# --- Environment Variable Configuration ---
Write-Host "Setting Ollama environment variables (User Scope)..."
# Use $env:USERPROFILE for user-specific settings
$OllamaBaseDir = $OllamaModelsDirectory # Base directory
$OllamaModelsPath = Join-Path $OllamaBaseDir "models" # Actual models subdir Ollama uses

# Create base directory if it doesn't exist
if (-not (Test-Path $OllamaBaseDir)) {
    New-Item -Path $OllamaBaseDir -ItemType Directory -Force | Out-Null
    Write-Host "Created Ollama base directory: $OllamaBaseDir"
}

# Set variables using setx for persistence across sessions (User scope)
try {
    setx OLLAMA_HOST "$($OllamaHost):$($OllamaPort)"
    setx OLLAMA_ORIGINS "$OllamaOrigins"
    setx OLLAMA_KEEP_ALIVE "$OllamaKeepAlive"
    setx OLLAMA_MODELS "$OllamaBaseDir" # Set the BASE directory [5, 19]
    setx OLLAMA_DEBUG (if ($SetOllamaDebug) { "1" } else { "0" })
    Write-Host "Ollama environment variables set for the current user." -ForegroundColor Green
    Write-Host "Changes will apply next time Ollama service/app starts."
} catch {
    Write-Warning "Failed to set environment variables using setx: $($_.Exception.Message)"
    Write-Warning "Manual configuration via System Properties -> Environment Variables may be needed."
}

# --- Restart Ollama Service/App (Best Effort) ---
# Ollama runs as a background process started by the tray app or `ollama serve`
# Restarting requires stopping the existing process and starting it again.
Write-Host "Attempting to restart Ollama background process..."
$ollamaProcesses = Get-Process -Name "ollama" -ErrorAction SilentlyContinue
if ($ollamaProcesses) {
    Write-Host "Stopping existing Ollama processes..."
    Stop-Process -Name "ollama" -Force -ErrorAction SilentlyContinue
    Start-Sleep -Seconds 3 # Allow time for process to terminate
}
# Start Ollama service/app - depends on how it was installed/run
# Option 1: If installed as service (less common now?)
# Restart-Service Ollama -ErrorAction SilentlyContinue
# Option 2: Start the background server via ollama serve
Write-Host "Starting Ollama server in the background..."
try {
    # Start ollama serve non-interactively. This might open a console window briefly.
    # A better way might be needed depending on Ollama's current behavior (e.g., using Task Scheduler or NSSM)
    Start-Process powershell.exe -ArgumentList "-NoProfile -WindowStyle Hidden -Command `"ollama serve`"" -Verb RunAs # Run elevated
```

```
to ensure permissions
    Start-Sleep -Seconds 5 # Give server time to start
} catch {
    Write-Warning "Could not automatically start 'ollama serve': $($_.Exception.Message). You may need to start it manually."
}

# --- Service Verification ---
Write-Host "Verifying Ollama service connectivity..."
$ollamaApiEndpoint = "http://$($OllamaHost):$($OllamaPort)"
try {
    $response = Invoke-WebRequest -Uri $ollamaApiEndpoint -UseBasicParsing -TimeoutSec 10
    if ($response.StatusCode -eq 200 -and $response.Content -like "*Ollama is running*") {
        Write-Host "Ollama service is running and accessible at $ollamaApiEndpoint" -ForegroundColor Green
    } else {
        Write-Warning "Ollama service might be running but returned unexpected content or status: $($response.StatusCode)"
    }
} catch {
    Write-Warning "Failed to connect to Ollama service at $ollamaApiEndpoint: $($_.Exception.Message)"
    Write-Warning "Ensure Ollama is running and check firewall settings if OLLAMA_HOST is not 127.0.0.1."
}

# --- Pull Initial Model ---
Write-Host "Pulling initial model: $OllamaModelToPull..."
try {
    ollama pull $OllamaModelToPull
    Write-Host "Model '$OllamaModelToPull' pulled successfully." -ForegroundColor Green
} catch {
    Write-Warning "Failed to pull model '$OllamaModelToPull': $($_.Exception.Message)"
}

# --- Create Custom Model with Global System Prompt ---
if (-not $SkipCustomModelCreation) {
    Write-Host "Creating custom model variant with embedded system prompt..."
    # Expand the knowledge path variable in the prompt template
    $ResolvedGlobalSystemPrompt = $GlobalSystemPromptTemplate -f $LocalKnowledgePath

    $modelfileContent = @"
FROM $OllamaModelToPull
SYSTEM """$ResolvedGlobalSystemPrompt"""
PARAMETER temperature 0.7 # Example: Set default parameters if desired
"@

    $modelfilePath = Join-Path $PSScriptRoot "Modelfile_Custom_$($OllamaModelToPull)"
    try {
        $modelfileContent | Out-File -FilePath $modelfilePath -Encoding UTF8 -Force
        $customModelName = "custom-$OllamaModelToPull"
        Write-Host "Building custom model '$customModelName'..."
        ollama create $customModelName -f $modelfilePath
        if ($LASTEXITCODE -eq 0) {
            Write-Host "Successfully created custom model '$customModelName'." -ForegroundColor Green
            Write-Host "Use this model name ('$customModelName') for interactions requiring the global prompt."
        } else {
```

```
            throw "Ollama create command failed."
        }
    } catch {
        Write-Warning "Failed to create custom model '$customModelName': $($_.Exception.Message)"
    } finally {
        if (Test-Path $modelfilePath) { Remove-Item $modelfilePath -Force }
    }
} else {
    Write-Host "Skipping custom model creation as requested."
}

# ================================================================================
# TOOL FUNCTION DEFINITIONS (Wrappers)
# ================================================================================
Write-Host "`n=== Defining Tool Wrapper Functions ===" -ForegroundColor Cyan

# --- Web Search Function ---
function Invoke-WebSearch {
    param(
        [Parameter(Mandatory=$true)]
        [string]$Query
    )
    Write-Verbose "Performing web search for: $Query"
    if ($SearchApiKey -eq "YOUR_SEARCH_API_KEY_HERE" -or -not $SearchApiKey) {
        return "Error: Web search requires a valid Search API Key configured in the script parameters (\$SearchApiKey)."
    }
    # Example using SearchAPI.io - adjust URI and parsing based on actual API docs
    $encodedQuery = [uri]::EscapeDataString($Query)
    $uri = "$SearchApiEndpoint`?q=$encodedQuery&engine=google" # Specify engine if needed

    try {
        # Note: SearchAPI.io might not require Auth header if key is in query param
        # $headers = @{ "Authorization" = "Bearer $SearchApiKey" }
        $response = Invoke-RestMethod -Uri $uri -Method Get -TimeoutSec 30
        # --- Parse the response ---
        # This part is HIGHLY dependent on the specific API's response structure
        if ($response.organic_results) {
            $results = $response.organic_results | Select-Object -First 5 | ForEach-Object {
                "Title: $($_.title)`nLink: $($_.link)`nSnippet: $($_.snippet)`n---"
            }
            return $results -join "`n`n"
        } elseif ($response.answer_box) {
             return "Answer Box: $($response.answer_box.answer)" # Or other relevant fields
        } else {
            return "Web search returned results, but format not recognized or no direct snippets found."
        }
    } catch {
        return "Error during web search for '$Query': $($_.Exception.Message)"
    }
}

# --- Command Execution Function ---
```

```powershell
function Execute-SystemCommand {
    param(
        [Parameter(Mandatory=$true)]
        [string]$Command
    )
    Write-Verbose "Executing system command: $Command"
    $ErrorActionPreference = 'Continue' # Allow capturing errors without stopping script
    $output = ""
    $exitCode = 0

    try {
        # Use Invoke-Command for better isolation and capturing of PS output/errors
        # This executes in a separate process, which is generally safer
        $scriptBlock =::Create($Command)
        $output = Invoke-Command -ScriptBlock $scriptBlock *>&1 | Out-String
        $exitCode = if ($?) { 0 } else { $LASTEXITCODE | 1 } # Get exit code if possible
    } catch {
        $output = "Error executing command '$Command': $($_.Exception.Message) `n $($_.ScriptStackTrace)"
        $exitCode = -1 # Indicate an exception occurred
    } finally {
         $ErrorActionPreference = 'Stop' # Restore preference
    }

    # Format output clearly for the LLM
    $result = @"
Command Executed: $Command
Exit Code: $exitCode
Output (stdout/stderr combined):
$($output.Trim())
"@
    return $result
}

# --- Browser Control Function (Basic Example) ---
function Open-UrlInChrome {
    param(
        [Parameter(Mandatory=$true)]
        [string]$Url
    )
    Write-Verbose "Opening URL in Chrome: $Url"
    try {
        # Simple version: just launch Chrome with the URL
        Start-Process chrome.exe $Url
        return "Successfully launched Chrome with URL: $Url"

        # Advanced version using Selenium (requires driver setup)
        # $Driver = Start-SeChrome -Headless # Or non-headless
        # Enter-SeUrl -Url $Url -Driver $Driver
        # # Perform more actions here...
        # Stop-SeDriver -Driver $Driver
        # return "Successfully opened $Url using Selenium-controlled Chrome."
```

```
    } catch {
        return "Error opening URL '$Url' in Chrome: $($_.Exception.Message)"
    }
}

# --- Placeholder for OS Interaction Wrappers ---
# Example: Get specific WMI info
function Get-ProcessorInfo {
    Write-Verbose "Getting processor information via WMI/CIM..."
    try {
        $procInfo = Get-CimInstance -ClassName Win32_Processor | Select-Object Name, NumberOfCores,
NumberOfLogicalProcessors, MaxClockSpeed
        return $procInfo | Out-String
    } catch {
        return "Error getting processor info: $($_.Exception.Message)"
    }
}

# Example: Wrapper for starting a service
function Start-WindowsService {
    param([string]$ServiceName)
    Write-Verbose "Attempting to start service: $ServiceName"
    return Execute-SystemCommand -Command "Start-Service -Name '$ServiceName'"
}


# ================================================================================
# IMPORTING CLONED FUNCTIONS
# ================================================================================
Write-Host "`n=== Importing Functions from Cloned Repository ===" -ForegroundColor Cyan

if (Test-Path $LocalFunctionsPath) {
    $functionFiles = Get-ChildItem -Path $LocalFunctionsPath -Filter *.ps1 -Recurse
    $importedCount = 0
    $failedCount = 0
    foreach ($file in $functionFiles) {
        try {
            . $file.FullName
            Write-Verbose "Sourced function file: $($file.Name)"
            $importedCount++
        } catch {
            Write-Warning "Failed to source function file '$($file.FullName)': $($_.Exception.Message)"
            $failedCount++
        }
    }
    Write-Host "Imported $importedCount function files (encountered $failedCount errors)."
} else {
    Write-Warning "Local functions path '$LocalFunctionsPath' not found. No functions imported."
}

# ================================================================================
# FINAL INSTRUCTIONS AND WRAP-UP
```

```powershell
# ================================================================================
Write-Host "`n=== Setup Complete ===" -ForegroundColor Green
Write-Host "Ollama is configured with integrated tools and knowledge base access."
Write-Host "Key Information:"
Write-Host " - Ollama API Endpoint: $ollamaApiEndpoint"
Write-Host " - Default Model Pulled: $OllamaModelToPull"
if (-not $SkipCustomModelCreation) { Write-Host " - Custom Model w/ Prompt: $customModelName (Recommended for use)" }
Write-Host " - PowerShell Functions Location: $LocalFunctionsPath"
Write-Host " - Knowledge Base Location: $LocalKnowledgePath"
Write-Host " - Ollama Models Directory: $OllamaBaseDir"
Write-Host "Available Tool Functions (usable via 'Execute-SystemCommand'):"
Write-Host "   - Invoke-WebSearch -Query <query>"
Write-Host "   - Execute-SystemCommand -Command <command_string>"
Write-Host "   - Open-UrlInChrome -Url <url>"
Write-Host "   - Get-ProcessorInfo"
Write-Host "   - Start-WindowsService -ServiceName <service>"
Write-Host "   - (Plus any functions sourced from $LocalFunctionsPath)"
Write-Host "   - Standard PowerShell cmdlets for Filesystem (Get-ChildItem, Get-Content, etc.)"
Write-Host "`nSecurity Reminder:" -ForegroundColor Yellow
Write-Host "This setup grants significant capabilities to the LLM. Monitor its activity closely."
Write-Host "Only run in trusted, isolated environments."
Write-Host "Consider enabling PowerShell Script Block Logging and Transcription for auditing."
Write-Host "To interact, use the Ollama CLI, API, or compatible clients, preferably targeting the '$customModelName' model."

# Example interaction function (Optional - can be used within the script or saved separately)
# function Invoke-OllamaChatWithTools {
#     param([string]$Prompt, [string]$Model = $customModelName) # Default to custom model
#     $ollamaApi = "$ollamaApiEndpoint/api/chat"
#     $body = @{
#         model = $Model
#         messages = @( @{ role = "user"; content = $Prompt } )
#         stream = $false # Keep it simple for this example
#     } | ConvertTo-Json -Depth 5

#     try {
#         $response = Invoke-RestMethod -Uri $ollamaApi -Method Post -Body $body -ContentType "application/json"
#         return $response.message.content
#     } catch {
#         Write-Error "Error communicating with Ollama API: $($_.Exception.Message)"
#         return $null
#     }
# }
# Example Usage:
# Invoke-OllamaChatWithTools -Prompt "Use Execute-SystemCommand to list running processes named 'chrome'."

Write-Host "`nScript execution finished."
```

6.1. Script Structure and Modules

The script is organized logically:

1. **Parameter Block:** Defines all configurable options.
2. **Pre-flight Checks:** Verifies administrator privileges and attempts self-elevation. Sets error preferences.
3. **Dependency Installation:** Installs Git and the Selenium PowerShell module if needed.
4. **Repository Cloning:** Clones the function and knowledge base repositories.

5. **Ollama Installation & Configuration:** Installs Ollama, configures PATH and environment variables, restarts the service, pulls the base model, and creates the custom model variant with the embedded system prompt.
6. **Tool Function Definitions:** Defines wrapper functions (Invoke-WebSearch, Execute-SystemCommand, Open-UrlInChrome, OS examples) that provide a cleaner interface for the LLM (though primarily invoked via Execute-SystemCommand in this setup).
7. **Importing Cloned Functions:** Dot-sources the .ps1 files from the cloned functions repository.
8. **Final Instructions:** Summarizes the setup and provides guidance for interaction and security.

6.2. Configuration Block Deep Dive

The param() block at the script's start is the central point for customization. Users *must* replace "YOUR_SEARCH_API_KEY_HERE" with a valid API key for web search to function.[20] Other parameters like repository URLs, Ollama settings ($OllamaHost, $OllamaPort, $OllamaModelsDirectory, $OllamaKeepAlive), the default model ($OllamaModelToPull), and the content of the $GlobalSystemPromptTemplate can be modified to suit specific requirements. The $OllamaModelsDirectory points to the *base* directory; Ollama typically creates a models subdirectory within it.[18]

6.3. Core Function Explanations

- Test-IsAdministrator: Checks if the script is running with elevated rights.[22]
- Clone-Repository: Handles the cloning logic, including the option to force a fresh clone.[9]
- Invoke-WebSearch: Sends a query to the configured Search API endpoint using Invoke-RestMethod and returns formatted results.[20] Requires $SearchApiKey.
- Execute-SystemCommand: The primary interface for the LLM to interact with the system. It takes a command string, executes it (using Invoke-Command for PowerShell or potentially cmd /c for external/batch commands for safety), captures combined standard output and error, and returns the result along with the exit code.[38] This function is crucial for enabling filesystem access, OS interaction, browser commands, and running functions from the cloned repository.
- Open-UrlInChrome: A simple wrapper using Start-Process to open a URL in the default Chrome browser. More complex interactions would require Selenium commands executed via Execute-SystemCommand.[13]
- OS Interaction Examples (Get-ProcessorInfo, Start-WindowsService): Demonstrate how specific OS tasks can be wrapped for clarity, although the LLM could achieve the same using Execute-SystemCommand with the underlying cmdlets (Get-CimInstance, Start-Service).[44]

7. Usage Instructions, Troubleshooting, and Final Recommendations

7.1. Running the Script

1. **Save:** Save the complete PowerShell script to a file named Setup-OllamaAgent.ps1.
2. **Configure:** Open the script in a text editor and carefully review the param() block. **Crucially, replace "YOUR_SEARCH_API_KEY_HERE" with your actual SearchAPI.io (or chosen provider's) API key.** Adjust other parameters (repositories, Ollama settings, prompt) as needed.
3. **Elevate:** Open a PowerShell terminal **as Administrator**. Right-click the PowerShell icon in the Start Menu and select "Run as administrator".[8]
4. **Navigate:** Change directory to where you saved the script: cd C:\path\to\script.

5. **Execute:** Run the script: .\Setup-OllamaAgent.ps1. If you modified parameters and want to override defaults, you can pass them: .\Setup-OllamaAgent.ps1 -OllamaModelToPull "mistral" -ForceReclone.
6. **Monitor:** Observe the script output for any warnings or errors during installation and configuration. Approve the UAC prompt if the script needed to self-elevate.

7.2. Post-Setup Interaction

Once the script completes successfully:

- **Target Model:** For interactions requiring the full context and capabilities defined in the system prompt, use the custom model created by the script (e.g., custom-llama3).
- **Interaction Methods:**
  - **Ollama CLI:** Open a terminal and use ollama run custom-llama3 (replace model name as needed).
  - **API Calls:** Send requests to the Ollama API endpoint (e.g., http://127.0.0.1:11434/api/chat or as configured) using tools like curl, Postman, or custom applications. Ensure the request body specifies the correct model name.
  - **Integrated Functions (via LLM):** Formulate prompts that instruct the LLM to use its tools via the Execute-SystemCommand function.
    - *Example 1 (Web Search & File):* "Search the web for 'latest news on Nvidia stock' using Invoke-WebSearch and save the first few results to a file named 'nvidia_news.txt' in my Documents folder." (The LLM would need to formulate Execute-SystemCommand -Command "Invoke-WebSearch -Query 'latest news on Nvidia stock' | Out-File -FilePath C:\Users\YourUser\Documents\nvidia_news.txt").
    - *Example 2 (Filesystem & Browser):* "List the files in my Downloads directory. Then, open Chrome to https://news.google.com." (LLM formulates Execute-SystemCommand -Command "Get-ChildItem -Path C:\Users\YourUser\Downloads" followed by Execute-SystemCommand -Command "Open-UrlInChrome -Url 'https://news.google.com'").
    - *Example 3 (OS Interaction):* "Check the status of the 'Print Spooler' service." (LLM formulates Execute-SystemCommand -Command "Get-Service -Name Spooler").
    - *Example 4 (Knowledge Base):* "Read the file '$LocalKnowledgePath\introduction.md' and tell me the project's goal." (LLM formulates Execute-SystemCommand -Command "Get-Content -Path '$LocalKnowledgePath\introduction.md'" - path needs resolving).

7.3. Troubleshooting Common Issues

- **Permission Errors:** Ensure the script is run from an elevated PowerShell prompt. Check permissions on target directories if file operations fail. UAC prompts must be accepted.[8]
- **Installation Failures:** Check internet connectivity. Ensure PowerShell Gallery is accessible (Register-PSRepository -Default). If Git/Ollama install fails, try manual installation first.[4] Antivirus software might interfere.[4]
- **Environment Variable Issues:** Changes might require restarting the Ollama service/app or even the terminal/system.[19] Verify variables using Get-ChildItem Env: in PowerShell or System Properties. Ensure setx commands completed without error.
- **Git Cloning Errors:** Verify repository URLs are correct and accessible. For private repos, Git credential management might be needed. Check network connectivity.
- **Selenium/ChromeDriver:** Ensure the Selenium module installed correctly. Browser automation might fail if the ChromeDriver version doesn't match the installed Chrome browser version. The module attempts automatic

download, but manual download/placement might be needed.[15] Check Selenium module documentation for driver path configuration.

- **Web Search API Key:** Ensure the $SearchApiKey is correct and the account is active. Check API provider documentation for usage limits and correct endpoint/authentication.[20]
- **Command Execution Failures:** Check command syntax. Commands requiring elevation might fail if the base Ollama process isn't running with sufficient rights (though Execute-SystemCommand attempts elevation via Invoke-Command or Start-Process -Verb RunAs where possible). Review output/errors returned by Execute-SystemCommand.
- **Ollama Service Unreachable:** Use curl http://<host>:<port> to test basic connectivity.[6] Check if the ollama process is running. Verify OLLAMA_HOST setting and firewall rules if accessing non-locally. Check Ollama logs for errors.[18]
- **Accessing Logs:** Ollama logs are typically found in %LOCALAPPDATA%\Ollama (app.log, server.log) and model/configuration files in %HOMEPATH%\.ollama (or the directory specified by $OLLAMA_MODELS).[18] Enable debug logging ($SetOllamaDebug = $true) for more details.

7.4. Security Best Practices Revisited

Given the extensive capabilities granted, security is paramount:

- **Isolation: Strongly recommended:** Run this entire setup within a dedicated, isolated Virtual Machine (VM) or container disconnected from sensitive networks or data. Do not run on primary workstations or production servers.
- **Monitoring:** Enable comprehensive PowerShell logging via Group Policy or local settings:
  - **Script Block Logging:** Captures the actual code blocks executed.[1] (Event ID 4104)
  - **Module Logging:** Logs pipeline execution details for specific modules. (Event ID 4103)
  - **Transcription:** Creates text logs of entire PowerShell sessions (Start-Transcript).[1] Regularly review these logs and Windows Event Logs (especially Process Creation events - ID 4688) for suspicious activity originating from the Ollama process or the script.[1]
- **Least Privilege:** Customize the script and the global system prompt. If certain capabilities (e.g., file deletion, specific cmdlets, browser control) are not needed, remove the corresponding functions, cmdlets from the prompt's description, and potentially add checks within Execute-SystemCommand to block forbidden commands.
- **Network Exposure:** Be extremely cautious when setting OLLAMA_HOST to 0.0.0.0 or a specific network IP.[5] Ensure appropriate firewall rules and authentication mechanisms (if available/implemented) are in place if exposing the Ollama API beyond localhost. Set OLLAMA_ORIGINS restrictively.[6]
- **Input Sanitization (Conceptual):** While difficult to enforce perfectly with an LLM, be aware that prompts could potentially be crafted to exploit command execution capabilities (prompt injection). The system prompt attempts to mitigate this by asking for clarification on ambiguous/harmful requests.
- **Regular Updates:** Keep Windows, PowerShell, Ollama, Git, and all models updated to patch potential security vulnerabilities.

7.5. Further Development and Exploration

This setup provides a powerful foundation. Further enhancements could include:

- **RAG Implementation:** Build a proper RAG pipeline (likely in Python using LangChain/LlamaIndex) to effectively utilize the cloned knowledge base, indexing the content and retrieving relevant chunks

semantically.[55] The RAG system could call the Ollama API hosted by this setup.

- **Advanced Agentic Frameworks:** Integrate this environment with agentic frameworks like LangGraph [61], AutoGen, or CrewAI. These frameworks can orchestrate complex tasks, manage memory, and implement reflection loops [58], using the tools provided by this PowerShell setup via API calls or potentially by executing the wrapper functions directly if bridging is implemented.
- **Refined Tool Wrappers:** Create more robust and specific PowerShell wrapper functions for common tasks (e.g., Get-SpecificRegistryValue, Create-ScheduledTask, more granular browser actions) instead of relying solely on Execute-SystemCommand for everything.
- **Structured Output:** Experiment with Ollama features or prompting techniques to request structured output (e.g., JSON) from commands or queries, making programmatic use of the results easier.
- **Model Exploration:** Test different Ollama models (Mistral, Phi, Gemma, etc.) to find the best balance of capability, speed, and resource consumption for specific tasks.[4]
- **Enhanced Security:** Implement more granular permission controls, potentially using Just Enough Administration (JEA) principles for the PowerShell endpoint [1], although integrating this directly with Ollama's execution model would be complex.

8. Conclusion

The PowerShell script and methodology outlined in this report successfully establish a highly integrated Ollama environment on Windows. By automating the installation of dependencies, cloning necessary resources, configuring Ollama settings, and providing functional interfaces for web search, filesystem manipulation, command execution, browser control, and OS interaction, it equips local LLMs with significant operational capabilities directly within the Windows ecosystem. The creation of a custom model variant ensures a consistent, capability-aware system prompt is applied.

However, the power derived from this deep integration necessitates extreme caution. The inherent security risks associated with granting an AI broad system access cannot be overstated. This solution is best suited for controlled, isolated research and development environments where rigorous monitoring and security practices are strictly enforced. Users must understand the implications and tailor the provided capabilities to the principle of least privilege for their specific use case.

This setup serves as a potent foundation for exploring advanced AI agent concepts on Windows, enabling experimentation with tool use, knowledge base interaction, and potentially facilitating more complex agentic behaviors like self-reflection when coupled with external orchestration frameworks.

Works cited

1. Securing PowerShell in the enterprise | Cyber.gov.au, accessed May 4, 2025, https://www.cyber.gov.au/resources-business-and-government/maintaining-devices-and-systems/system-hardening-and-administration/system-administration/securing-powershell-enterprise
2. Detecting PowerShell Exploitation - eG Innovations, accessed May 4, 2025, https://www.eginnovations.com/blog/detecting-powershell-exploitation/
3. Is running Powershell always as administrator a good practice?, accessed May 4, 2025, https://security.stackexchange.com/questions/225773/is-running-powershell-always-as-administrator-a-good-practice

4. How to Install Ollama on Windows: A Step-by-Step Guide - BytePlus, accessed May 4, 2025, https://www.byteplus.com/en/topic/398049
5. Global Configuration Variables for Ollama · Issue #2941 - GitHub, accessed May 4, 2025, https://github.com/ollama/ollama/issues/2941
6. Set up Ollama on Windows | GPT for Work Documentation, accessed May 4, 2025, https://gptforwork.com/help/ai-models/local-servers/set-up-ollama-on-windows
7. Set up Ollama on Windows | GPT for Work Documentation, accessed May 4, 2025, https://gptforwork.com/help/ai-models/ollama/ollama-setup-windows
8. Adding a User to the Administrators Group Using PowerShell - IT Master Services, accessed May 4, 2025, https://www.itms-us.com/Tips-And-Tricks/Adding-A-User-To-The-Administrators-Group-Using-PowerShell
9. 2.1 Git Basics - Getting a Git Repository, accessed May 4, 2025, https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository
10. Clone an existing Git repo - Azure Repos | Microsoft Learn, accessed May 4, 2025, https://learn.microsoft.com/en-us/azure/devops/repos/git/clone?view=azure-devops
11. Cloning a repository - GitHub Docs, accessed May 4, 2025, https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository
12. A Detailed Guide on How to use Selenium With PowerShell | BrowserStack, accessed May 4, 2025, https://www.browserstack.com/guide/how-to-use-selenium-with-powershell
13. PowerShell module to run a Selenium WebDriver. - GitHub, accessed May 4, 2025, https://github.com/adamdriscoll/selenium-powershell
14. StepAutomation 1.0.5 - PowerShell Gallery, accessed May 4, 2025, https://www.powershellgallery.com/packages/StepAutomation/1.0.5
15. How to Use Selenium with PowerShell [Tutorial 2025] - Scrapeless, accessed May 4, 2025, https://www.scrapeless.com/en/blog/selenium-powershell
16. How to Use Selenium With PowerShell [Tutorial 2025] - ZenRows, accessed May 4, 2025, https://www.zenrows.com/blog/selenium-powershell
17. Ollama agent README - PowerShell | Microsoft Learn, accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/utility-modules/aishell/developer/ollama-agent-readme?view=ps-modules
18. Ollama Windows Config Guide | Restackio, accessed May 4, 2025, https://www.restack.io/p/ollama-answer-windows-config-cat-ai
19. Change Ollama Models Directory: A Complete Guide - BytePlus, accessed May 4, 2025, https://www.byteplus.com/en/topic/418090
20. Ollama Function Calling with Search API - GitHub, accessed May 4, 2025, https://github.com/srbhr/Ollama-function-calling
21. An introduction to function calling and tool use - Apideck, accessed May 4, 2025, https://www.apideck.com/blog/llm-tool-use-and-function-calling
22. How to Run PowerShell Script as Administrator? - SharePoint Diary, accessed May

4, 2025, https://www.sharepointdiary.com/2015/01/run-powershell-script-as-administrator-automatically.html
23. how to run a powershell script as administrator - Super User, accessed May 4, 2025, https://superuser.com/questions/108207/how-to-run-a-powershell-script-as-administrator
24. How to Open an Elevated PowerShell Prompt in Windows 10/11 | NinjaOne, accessed May 4, 2025, https://www.ninjaone.com/blog/open-an-elevated-powershell-prompt/
25. about_Execution_Policies - PowerShell - Learn Microsoft, accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-7.5
26. A beginner's guide to Git and GitHub using PowerShell | PDQ, accessed May 4, 2025, https://www.pdq.com/blog/a-beginners-guide-to-git-and-github-using-powershell/
27. How to Call a PowerShell Function from another PS1 File? - SharePoint Diary, accessed May 4, 2025, https://www.sharepointdiary.com/2020/11/how-to-call-powershell-function-from-ps1-file.html
28. Script modules - PowerShell | Microsoft Learn, accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/scripting/learn/ps101/10-script-modules?view=powershell-7.5
29. How to Reuse Windows PowerShell Functions in Scripts - Microsoft Developer Blogs, accessed May 4, 2025, https://devblogs.microsoft.com/scripting/how-to-reuse-windows-powershell-functions-in-scripts/
30. PowerShell Import-Module with .ps1 quirk. - GitHub Gist, accessed May 4, 2025, https://gist.github.com/magnetikonline/2cdbfe45258c0cc3cf1530548baf30a7
31. Ollama Change System Prompt | Restackio, accessed May 4, 2025, https://www.restack.io/p/ollama-answer-change-system-prompt-cat-ai
32. Ollama Installation for macOS, Linux, and Windows - GitHub Pages, accessed May 4, 2025, https://translucentcomputing.github.io/kubert-assistant-lite/ollama.html
33. Ollama Set System Prompt | Restackio, accessed May 4, 2025, https://www.restack.io/p/ollama-answer-set-system-prompt-cat-ai
34. Can't overwrite default system prompt with /api/chat without creating a new model #8729, accessed May 4, 2025, https://github.com/ollama/ollama/issues/8729
35. Working with files and folders - PowerShell | Microsoft Learn, accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/scripting/samples/working-with-files-and-folders?view=powershell-7.5
36. about_FileSystem_Provider - PowerShell | Microsoft Learn, accessed May 4, 2025,

https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_filesystem_provider?view=powershell-7.5
37. Access network share without mapping drive letter (PowerShell) - Server Fault, accessed May 4, 2025, https://serverfault.com/questions/549040/access-network-share-without-mapping-drive-letter-powershell
38. How do I capture the output into a variable from an external process in PowerShell?, accessed May 4, 2025, https://stackoverflow.com/questions/8097354/how-do-i-capture-the-output-into-a-variable-from-an-external-process-in-powershe
39. Powershell - how to log all external cmd output? - Server Fault, accessed May 4, 2025, https://serverfault.com/questions/765953/powershell-how-to-log-all-external-cmd-output
40. Capturing Output - Windows PowerShell Quick Reference [Book] - O'Reilly Media, accessed May 4, 2025, https://www.oreilly.com/library/view/windows-powershell-quick/0596528132/ch01s20.html
41. ChromeDevTools/awesome-chrome-devtools: Awesome tooling and resources in the Chrome DevTools & DevTools Protocol ecosystem - GitHub, accessed May 4, 2025, https://github.com/ChromeDevTools/awesome-chrome-devtools
42. Chrome DevTools Protocol - GitHub Pages, accessed May 4, 2025, https://chromedevtools.github.io/devtools-protocol/
43. Use the Chrome DevTools Protocol (CDP) in WebView2 apps - Learn Microsoft, accessed May 4, 2025, https://learn.microsoft.com/en-us/microsoft-edge/webview2/how-to/chromium-devtools-protocol
44. How to Start or Stop Windows Service Remotely on Several PCs? - Action1, accessed May 4, 2025, https://www.action1.com/blog/how-to-start-stop-windows-service-remotely-on-several-pcs/
45. Set-Service (Microsoft.PowerShell.Management), accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/set-service?view=powershell-7.5
46. Start-Service (Microsoft.PowerShell.Management), accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/module/microsoft.powershell.management/start-service?view=powershell-7.5
47. Working with WMI - PowerShell | Microsoft Learn, accessed May 4, 2025, https://learn.microsoft.com/en-us/powershell/scripting/learn/ps101/07-working-with-wmi?view=powershell-7.5
48. WMI Commands - powershell.one, accessed May 4, 2025, https://powershell.one/wmi/commands
49. WMI Reference for PowerShell, accessed May 4, 2025, https://powershell.one/wmi/
50. Creating .NET and COM objects - PowerShell - Learn Microsoft, accessed May 4,

2025, https://learn.microsoft.com/en-us/powershell/scripting/samples/creating-.net-and-com-objects--new-object-?view=powershell-7.5
51. Using COM with Windows PowerShell 1.0 - Techotopia, accessed May 4, 2025, https://www.techotopia.com/index.php/Using_COM_with_Windows_PowerShell_1.0
52. Reinventing PowerShell in C/C++ - itm4n's blog, accessed May 4, 2025, https://itm4n.github.io/reinventing-powershell/
53. pinvoke | Learn Powershell | Achieve More, accessed May 4, 2025, https://learn-powershell.net/tag/pinvoke/
54. Use PowerShell to Interact with the Windows API: Part 1 - Scripting Blog [archived], accessed May 4, 2025, https://devblogs.microsoft.com/scripting/use-powershell-to-interact-with-the-windows-api-part-1/
55. Transforming literature screening: The emerging role of large language models in systematic reviews - PMC - PubMed Central, accessed May 4, 2025, https://pmc.ncbi.nlm.nih.gov/articles/PMC11745399/
56. Addressing Asymmetry in Contrastive Learning: LLM-Driven Sentence Embeddings with Ranking and Label Smoothing - MDPI, accessed May 4, 2025, https://www.mdpi.com/2073-8994/17/5/646
57. Agents - LlamaIndex, accessed May 4, 2025, https://docs.llamaindex.ai/en/stable/module_guides/deploying/agents/
58. A Guide to Reflection Agents Using LlamaIndex - Analytics Vidhya, accessed May 4, 2025, https://www.analyticsvidhya.com/blog/2024/08/a-guide-to-reflection-agents-using-llamaindex/
59. Agentic Workflows for Improving LLM Reasoning in Robotic Object-Centered Planning, accessed May 4, 2025, https://www.preprints.org/manuscript/202501.0131/v1
60. Reflexion | Prompt Engineering Guide, accessed May 4, 2025, https://www.promptingguide.ai/techniques/reflexion
61. Reflection Agents - LangChain Blog, accessed May 4, 2025, https://blog.langchain.dev/reflection-agents/
62. #12: How Do Agents Learn from Their Own Mistakes? The Role of Reflection in AI, accessed May 4, 2025, https://huggingface.co/blog/Kseniase/reflection
63. #12: How Do Agents Learn from Their Own Mistakes? The Role of Reflection in AI, accessed May 4, 2025, https://www.turingpost.com/p/reflection
64. What is Agentic AI Reflection Pattern? - Analytics Vidhya, accessed May 4, 2025, https://www.analyticsvidhya.com/blog/2024/10/agentic-ai-reflection-pattern/
65. Asking Tiny Questions: Local LLMs for PowerShell Devs - Chrissy LeMaire, accessed May 4, 2025, https://blog.netnerds.net/2025/03/asking-tiny-questions-powershell-local-llm/