# Unity Automation — Handover Manual (All-in-One)

Generated: 2025-08-20T05:38:35.171738 UTC

## Overview

This document bundles everything a new assistant (new chat) needs to pick up the Unity automation project. It contains a detailed description of two workflows, the intended relationship between them, the scripts/commands used, and the exact PowerShell router script (detect-only/latest) so the next assistant can reproduce and continue work without loss of context.

## Workflows (naming and purpose)

Important naming: we name the \*Git & Patch Router\* system as \*\*Workflow A\*\*, and the \*Realtime Voice-driven Unity\* system as \*\*Workflow B\*\*.

Why: Workflow A is used to create, deliver, and maintain Workflow B. Workflow B is the runtime environment inside Unity where you talk to the assistant by voice and the assistant edits the Unity project in-play. Keeping this naming convention consistent is crucial when handing to a new chat/assistant.

### Workflow A — Git + Patch Router (producer of patches)

Purpose: Observe the Unity project repository, accept patches produced by assistants, and apply them automatically so the Unity project is kept up-to-date. Workflow A runs outside Unity and is resilient to Unity crashes: it stores, verifies, and applies patches using git and a PowerShell inbox-router.

Key components:

- Git repository (project root).

- inbox-router.ps1 (PowerShell script) which watches Downloads and applies patches automatically or in-place depending on mode.

- Logs: patch-router.log and patch-inbox.log stored in the user's profile folder (see paths in scripts).

Core git operations used by the router:

- `git apply --check --3way --whitespace=nowarn <patch>` (verify patch)

- `git apply --index --3way --whitespace=nowarn <patch>` (apply and stage)

- `git add Assets -A` (ensure .meta files are included)

- `git commit -m 'inbox: apply <patch>'`

- `git push`

### Workflow B — Realtime Voice Assistant inside Unity (consumer and testbed)

Purpose: This is the in-Unity voice assistant. While running in Play Mode the user speaks to the assistant and instructs it to design or modify gameplay, scenes, or systems (for example: 'Create a 3D Snake game'). The assistant performs edits (via scripted operations or by generating patches applied by Workflow A), the user tests in Play Mode, and then exits Play Mode to commit/log results.

Key guarantees & behavior:

- The assistant running inside Unity (Workflow B) is \*stateful\* during Play Mode, and can run code that manipulates the scene/project.

- When the user exits Play Mode, Workflow B should produce:

\* A concise log of actions taken (timestamped).

\* Any generated or modified files staged in the Unity project (or exported as a patch).

- Workflow A consumes these logs/patches (either pushed directly, or produced by the assistant and downloaded) and applies them to the repository so the next test loop starts from the updated project.

## High-level Loop (how A & B work together)

1. Developer (you) run Unity and enter Play Mode. The voice assistant inside Unity (Workflow B) listens and edits live.

2. While testing in Play Mode, the assistant may create or propose changes; these may be output as patches or recorded as logs.

3. When you exit Play Mode, Workflow B saves logs and either commits to git or produces a patch file and notifies you (the developer) in the chat.

4. You paste the patch into the chat or the assistant places a downloadable patch link in the chat. You download the patch locally.

5. The inbox-router (Workflow A) watches Downloads, detects the final `.patch` file as soon as the browser finishes downloading it, and instantly applies it (or applies in-place) with `git apply` / commit / push.

6. You reopen Play Mode to test the applied changes. Repeat.

## How a new chat assistant should pick up where we left off

Copy this document into the new chat (or upload the DOCX) and then ask the new assistant to:

1. Verify the repo URL, branches, and credentials so the assistant can inspect the repository.

2. Inspect `patch-router.log` and `patch-inbox.log` (paths referenced in this document) to understand recent activity.

3. Confirm the `inbox-router.ps1` content (the PowerShell router) and the scheduled task name `UnityPatchRouter`. If changes are needed, update the script and restart the task.

4. Request that the human developer provides the latest patch number (the single most important piece of information) before the assistant generates a new patch. The assistant should rely \*only\* on the last patch number the human mentions when deciding what patch to send next — never assume it hasn't been downloaded.

## Essential commands & scripts (copy/paste)

PowerShell: start/stop the router scheduled task (used on Windows):

```  
schtasks /End /TN "UnityPatchRouter" 2>$null  
schtasks /Run /TN "UnityPatchRouter"  
```

Tail the router log:

```  
Get-Content "$env:USERPROFILE\patch-router.log" -Wait  
```

Manually run the inbox-router script (for debugging):

```  
& "$env:USERPROFILE\inbox-router.ps1"  
```

Git apply/check commands used by the router (the router runs these):

```  
git apply --check --3way --whitespace=nowarn "<patchfile>"  
git apply --index --3way --whitespace=nowarn "<patchfile>"  
git add Assets -A  
git commit -m "inbox: apply <patchfile>"  
git push  
```

## Appendix A — inbox-router.ps1 (detect-only v1.0.3)

The exact script below is the most recent 'detect-only' router that watches Downloads and logs READY events. It is saved to the user's profile at `inbox-router.ps1`. Paste & run it in PowerShell to reproduce the environment.

# router-detect-only v1.0.3 (2025-08-19) -- detect-only variant (no apply/move)  
param(  
 [string]$Project = "C:\Users\ander\My project", # unused (detect-only)  
 [string]$LogPath = "$env:USERPROFILE\patch-router.log"  
)  
  
$SCRIPT\_VERSION = "router-detect-only v1.0.3 (2025-08-19)"  
$ErrorActionPreference = "SilentlyContinue"  
function Write-Log($m){ "[$(Get-Date -Format s)] $m" | Out-File -Append $LogPath }  
  
function Wait-FileReady($Path, $TimeoutSec=120){  
 $sw=[Diagnostics.Stopwatch]::StartNew()  
 while($sw.Elapsed.TotalSeconds -lt $TimeoutSec){  
 if(Test-Path $Path){  
 try{  
 $a=(Get-Item $Path).Length; Start-Sleep -Milliseconds 300  
 $b=(Get-Item $Path).Length; if($a -eq $b){ return $true }  
 }catch{}  
 }  
 Start-Sleep -Milliseconds 200  
 }  
 return $false  
}  
  
# Boot banner + version control  
try{  
 $scriptPath = $MyInvocation.MyCommand.Path  
 $sha = if (Test-Path $scriptPath) { (Get-FileHash $scriptPath -Algorithm SHA256).Hash } else { "<no-path>" }  
 $psver = $PSVersionTable.PSVersion  
 $arch = ( [IntPtr]::Size -eq 8 ) ? '64' : '32'  
 Write-Log ("BOOT who={0} pid={1} pwsh={2} {3}-bit cwd={4}" -f [System.Security.Principal.WindowsIdentity]::GetCurrent().Name, $PID, $psver, $arch, (Get-Location))  
 Write-Log ("SCRIPT version={0} path={1} sha256={2}" -f $SCRIPT\_VERSION, $scriptPath, $sha)  
}catch{  
 Write-Log ("BOOT banner exception: {0}" -f $\_.Exception.Message)  
}  
  
# Downloads discovery (robust)  
function Get-DownloadFolders {  
 $c=@()  
 try{  
 $guid = '{374DE290-123F-4565-9164-39C4925E467B}'  
 $rk = 'HKCU:\SOFTWARE\Microsoft\Windows\CurrentVersion\Explorer\User Shell Folders'  
 $val = (Get-ItemProperty -Path $rk -ErrorAction Stop).$guid  
 if($val){  
 $expanded = [Environment]::ExpandEnvironmentVariables($val)  
 if(Test-Path $expanded){ $c += $expanded }  
 }  
 }catch{}  
 $c += "$env:USERPROFILE\Downloads"  
 $c += "$env:USERPROFILE\OneDrive\Downloads"  
 foreach($root in @("$env:LOCALAPPDATA\Google\Chrome\User Data", "$env:LOCALAPPDATA\Microsoft\Edge\User Data")){  
 if(Test-Path $root){  
 Get-ChildItem -Path $root -Directory -ErrorAction SilentlyContinue | ForEach-Object {  
 $prefs = Join-Path $\_.FullName "Preferences"  
 if(Test-Path $prefs){  
 try{  
 $json = Get-Content -Raw $prefs | ConvertFrom-Json  
 $dir = $json.download.default\_directory  
 if($dir -and (Test-Path $dir)){ $c += $dir }  
 }catch{}  
 }  
 }  
 }  
 }  
 $c | Where-Object { $\_ -and (Test-Path $\_) } | Select-Object -Unique  
}  
  
# Debounced event -> readiness probe  
$script:debounce = @{}  
$script:watchers = New-Object System.Collections.ArrayList  
$script:subs = New-Object System.Collections.ArrayList  
  
function Queue-Probe([string]$Path,[string]$Source){  
 if(-not $Path){ return }  
 Write-Log ("EVT {0} name='{1}' full='{2}'" -f $Source, (Split-Path -Leaf $Path), $Path)  
  
 # Only care about .patch / .patch.crdownload for readiness  
 $low = $Path.ToLower()  
 if( ($low -notlike "\*.patch") -and ($low -notlike "\*.patch.crdownload") ){ return }  
  
 # Normalize to final .patch path (strip .crdownload)  
 $final = if($low -like "\*.patch.crdownload"){ $Path -replace '\.crdownload$','' } else { $Path }  
 try { $final = [System.IO.Path]::GetFullPath($final) } catch {}  
 $key = $final.ToLower()  
  
 # Debounce 1.2s  
 if($script:debounce.ContainsKey($key)){  
 try{ $script:debounce[$key].Stop(); $script:debounce[$key].Start() }catch{}  
 return  
 }  
  
 $t = New-Object System.Timers.Timer  
 $t.Interval = 1200  
 $t.AutoReset = $false  
 $null = Register-ObjectEvent -InputObject $t -EventName Elapsed -MessageData $final -Action {  
 $target = $Event.MessageData  
 try{  
 $sw=[Diagnostics.Stopwatch]::StartNew()  
 while($sw.Elapsed.TotalSeconds -lt 120){  
 if(Test-Path $target){  
 try{  
 $a=(Get-Item $target).Length; Start-Sleep -Milliseconds 300  
 $b=(Get-Item $target).Length  
 if($a -eq $b){ break }  
 }catch{}  
 }  
 Start-Sleep -Milliseconds 200  
 }  
 if(-not (Test-Path $target)){  
 Write-Log ("READY miss: final disappeared '{0}'" -f $target); return  
 }  
 $fi = Get-Item $target  
 Write-Log ("READY detected: '{0}' size={1} lastWriteUtc={2:o}" -f $fi.FullName, $fi.Length, $fi.LastWriteTimeUtc)  
 # (detect-only: no apply/move)  
 }catch{  
 Write-Log ("READY exception for '{0}': {1}" -f $target, $\_.Exception.Message)  
 }  
 }  
  
 $script:debounce[$key] = $t  
 $t.Start()  
}  
  
# Wire watchers + self-test  
$folders = Get-DownloadFolders  
foreach ($dir in $folders) { Write-Log ("DIR candidate: {0} exists={1}" -f $dir, (Test-Path $dir)) }  
  
foreach ($dir in $folders) {  
 try{  
 $fsw = New-Object System.IO.FileSystemWatcher  
 $fsw.Path = $dir  
 $fsw.Filter = "\*"  
 $fsw.NotifyFilter = [IO.NotifyFilters]'FileName, LastWrite, Size'  
 $fsw.IncludeSubdirectories = $false  
 $fsw.InternalBufferSize = 65536  
 $fsw.EnableRaisingEvents = $true  
  
 $sub1 = Register-ObjectEvent $fsw Created -Action { $p = $EventArgs.FullPath; if(-not $p -and $Event.SourceEventArgs){ $p = $Event.SourceEventArgs.FullPath }; if($p){ Queue-Probe $p "Created" } else { Write-Log "EVT Created missing FullPath" } }  
 $sub2 = Register-ObjectEvent $fsw Renamed -Action { $p = $EventArgs.FullPath; if(-not $p -and $Event.SourceEventArgs){ $p = $Event.SourceEventArgs.FullPath }; if($p){ Queue-Probe $p "Renamed" } else { Write-Log "EVT Renamed missing FullPath" } }  
 $sub3 = Register-ObjectEvent $fsw Changed -Action { $p = $EventArgs.FullPath; if(-not $p -and $Event.SourceEventArgs){ $p = $Event.SourceEventArgs.FullPath }; if($p){ Queue-Probe $p "Changed" } else { Write-Log "EVT Changed missing FullPath" } }  
 $sub4 = Register-ObjectEvent $fsw Deleted -Action { $p = $EventArgs.FullPath; if(-not $p -and $Event.SourceEventArgs){ $p = $Event.SourceEventArgs.FullPath }; Write-Log ("EVT Deleted name='{0}' full='{1}'" -f (Split-Path -Leaf $p), $p) }  
 $sub5 = Register-ObjectEvent $fsw Error -Action { try{ $msg = $EventArgs.GetException().Message }catch{ $msg = "<no ex>" }; Write-Log ("WATCH error: {0}" -f $msg) }  
  
 $null = $script:watchers.Add($fsw)  
 $null = $script:subs.Add($sub1)  
 $null = $script:subs.Add($sub2)  
 $null = $script:subs.Add($sub3)  
 $null = $script:subs.Add($sub4)  
 $null = $script:subs.Add($sub5)  
  
 Write-Log ("WATCH added: {0} subs+={1} totalWatchers={2} totalSubs={3} buf={4}" -f $dir, 5, $script:watchers.Count, $script:subs.Count, $fsw.InternalBufferSize)  
  
 # self-test: synthetic event (create & delete a temp file) - proves events are flowing  
 try{  
 $probe = Join-Path $dir ("\_fsw\_probe\_{0}.tmp" -f $PID)  
 Set-Content -Path $probe -Value ("probe {0} {1:o}" -f $PID, (Get-Date).ToUniversalTime()) -Encoding ascii  
 Start-Sleep -Milliseconds 100  
 Remove-Item -LiteralPath $probe -Force -ErrorAction SilentlyContinue  
 Write-Log ("PROBE wrote+deleted: {0}" -f $probe)  
 }catch{  
 Write-Log ("PROBE exception: {0}" -f $\_.Exception.Message)  
 }  
 }catch{  
 Write-Log ("WATCHER failed '{0}': {1}" -f $dir, $\_.Exception.Message)  
 }  
}  
  
Write-Log ("ROUTER active (detect-only) v={0} watchers={1} subs={2}" -f $SCRIPT\_VERSION, $script:watchers.Count, $script:subs.Count)  
  
# periodic heartbeat so we can see which instance owns the file  
Start-Job -ScriptBlock {  
 param($LogPath,$SCRIPT\_VERSION)  
 while($true){  
 "[$(Get-Date -Format s)] HEARTBEAT pid=$PID v=$SCRIPT\_VERSION" | Out-File -Append $LogPath  
 Start-Sleep -Seconds 15  
 }  
} -ArgumentList $LogPath,$SCRIPT\_VERSION | Out-Null  
  
# Keep alive  
while($true){ Start-Sleep -Seconds 3600 }

## Troubleshooting & Tips

- If a patch isn't being detected: check Downloads path, check patch-router.log for READY events, and confirm the router scheduled task is running.

- Always tell the assistant the \*last\* patch number you downloaded. The assistant should never assume older patches are relevant.

- If Unity crashes: Workflow A still holds the repository state. Use the git history to rollback or inspect recent commits.

- When switching to a new chat, upload this DOCX so the new assistant has the exact script, logs locations, and workflow goals.

## Document version & checksum

This document file checksum (SHA256) will be provided after save. Use this to ensure the new chat has the exact same copy.