ChatGPT

Controlling DaVinci Resolve from WSL2: Approaches and Examples

WSL-to-Windows Automation Patterns

Running DaVinci Resolve on Windows 11 while orchestrating it from WSL2 (Ubuntu) requires bridging the Linux environment (logic) with Windows (execution). A common pattern is to treat WSL as the high-level controller and use interop calls or network APIs to trigger actions on Windows. For example, the open-source **DaVinci Resolve MCP** project detects if it's running under WSL and then *launches its control server on Windows via PowerShell* (using .NET's `ProcessStartInfo` to handle I/O) [1] [2]. In practice, this means a WSL script can spawn a Windows process (e.g. a Python script or automation agent) that interacts with Resolve. This "WSL as brain, Windows as hands" architecture isolates the automation logic in Linux while executing actual GUI/API commands natively on Windows. It's essentially an RPC pattern – WSL issues commands, and a Windows-side component carries them out.

Key techniques for bridging WSL and Windows include: calling Windows executables from WSL (e.g. using `powershell.exe` or `cmd.exe` via WSL's interoperability), running local network services, or using shared resources (like a file or socket) for communication. The MCP project above uses PowerShell to start the Resolve automation server in Windows and relay input/outputs across the WSL boundary [3] [4]. Another approach is to run a **server inside Resolve** itself that WSL can talk to – for instance, a REST API service. The **davinci-rest** project does this by launching a FastAPI server from within Resolve's scripting environment, exposing endpoints to create projects, import media, add timeline clips, etc., all accessible via `http://localhost:5001` [5] [6]. In both cases, WSL can act as the client (making HTTP requests or launching PowerShell commands) while the heavy lifting happens on the Windows side where Resolve runs.

Using Resolve's Scripting API via Bridges

Blackmagic provides an official **DaVinci Resolve scripting API** (accessible through Python or Lua) for automating many editor functions (project management, timeline edits, color grading controls, etc.). Normally, scripts using this API must run on the host OS with Resolve (loading the `fusionscript` DLL on Windows). Projects have emerged to make this accessible from outside. For example, **DaVinci Resolve MCP** acts as a middleman between AI or external scripts and Resolve – it connects to Resolve's API and listens for commands (e.g. from a chatGPT-like assistant) [7]. Similarly, **davinci-rest** (mentioned above) effectively turns Resolve into a local web server with a Python API client [8] [9]. These allow high-level operations (like *"open project X"* or *"add a clip to timeline"*) to be triggered remotely.

Such API-centric automation is robust for tasks that the API covers. You can create bins, import media, set in/out points, or even manipulate color nodes through documented calls instead of mimicking clicks. A real-world example is **CommandPost** (on macOS), which uses Lua scripting and the Resolve API to automate editing workflows. On Windows/Linux, Python frameworks like **pydavinci** provide a more Pythonic interface to the same API [10]. In practice, you might run a Python script (or a persistent server) on Windows that

loads `DaVinciResolveScript` and listens for instructions from WSL (via a socket or REST). The benefit is avoiding fragile UI interactions in favor of official calls. The limitation is that **not every UI action is exposed** – for instance, adjusting certain tool parameters or triggering GUI-only features might not have a direct API method.

Keyboard Shortcuts and GUI Macro Tools

To fully automate Resolve "as if operated by a human," one often resorts to simulating keyboard and mouse input. This can be done from WSL by invoking Windows automation utilities:

- **PowerShell + .NET:** PowerShell can leverage .NET frameworks to send input to Windows applications. A script can find the Resolve window and bring it to the foreground, then dispatch keystrokes. For example, the MCP project builds a PowerShell snippet that gets the Resolve process, calls the Win32 `SetForegroundWindow` API on it, waits briefly, then uses `.NET`'s **SendKeys** to simulate a keystroke [11] [12]. This technique can be triggered from WSL by running `powershell.exe -Command ...` with the appropriate script. In essence, your WSL bash script can call out to PowerShell to push **Ctrl+S** or **F12** in Resolve just as a user would. (It's equivalent to using VBScript's `WScript.Shell.SendKeys`; indeed, one can also create a WScript COM object in PowerShell to achieve the same [13].) The key is to ensure the correct window is active and ready – the script above, for instance, includes a 200 ms sleep after focusing the Resolve window to avoid sending keys too early [14] [15].

- **AutoHotkey (AHK):** AHK is a popular Windows automation tool capable of complex macros. It's often recommended for automating creative apps. Blackmagic forum users have suggested AutoHotkey for repetitive Resolve tasks [16], and community members share extensive AHK scripts for Resolve (for example, automating timeline edits, reseting viewer zoom, etc.). An AHK script can be run from WSL by invoking `AutoHotkey.exe` with the script path (since WSL can launch Windows executables). AHK can send keystrokes, mouse clicks, or even use UI Automation to select menu items. Unlike simple SendKeys, AHK gives fine control: you can wait for specific window titles, coordinate clicks relative to UI elements, or respond to images on screen. Real-world case: some editors use MIDI controllers mapped via AutoHotkey to drive Resolve [17] – pressing a hardware button triggers AHK to execute a series of shortcuts in Resolve. This underscores that with the right scripting, AHK can make Resolve respond to virtually any external trigger.

- **UI Automation Frameworks / WinAppDriver:** For a more structured approach, **WinAppDriver** (Windows Application Driver) can automate UI elements via the WebDriver (Selenium) protocol. You run the WinAppDriver service on Windows, then from WSL you could use an Appium client or Selenium to locate buttons and press them. WinAppDriver acts as a server that accepts JSON commands to click, type, etc., on UI controls [18] [19]. Tools like the WinAppDriver UI Recorder or RobotFramework's WADLibrary build on this to help identify UI element *XPath* or selectors. In theory, one could script "click the 'Export' button" by its automation ID instead of relying on screen coordinates. **However,** DaVinci Resolve's UI is built with Qt, which doesn't expose much accessibility data to Windows UI Automation. Developers report that most controls show up as generic "Custom" widgets with no names or values via tools like Inspect or pywinauto [20]. This means that pure UIAutomation scripts might not read the internal UI state (e.g. list of clips or node properties) easily. WinAppDriver can still send low-level inputs (key presses) or perhaps interact with top-level windows and dialogs, but it may struggle to identify sub-components of Resolve's interface. In practice, many

automation scripters fall back to keyboard shortcuts and relative mouse clicks for Resolve, given these limitations.

- **AutoIt and Others:** AutoIt is similar to AHK (a BASIC-like scripting for Windows GUI) and could be used as well. Microsoft's **Power Automate Desktop** is another option – it provides a GUI to record actions and automate apps (essentially an RPA tool). These aren't commonly cited in Resolve-specific contexts, but the concept is the same: you could record a macro of UI actions and replay it. The choice often comes down to familiarity and the complexity of tasks – simple tasks might just use SendKeys via PowerShell, whereas complex workflows (with conditional logic or image recognition) lean toward AHK or a full automation framework.

Example Projects and Case Studies

- **DaVinci Resolve MCP:** This open-source project (by Samuel G.) is a comprehensive case study in Resolve automation. It runs a local server that accepts high-level commands (often from AI assistants like Cursor or Claude) and executes them in Resolve. Under the hood it uses *both* the official Resolve API and input simulation. For instance, to implement certain editing actions, MCP will programmatically call Resolve's Python API, but for something like triggering a specific UI button or shortcut, it falls back on sending a key combo via PowerShell (supported on both Windows and WSL setups) [21] [22]. The MCP architecture uses WSL as a controller in some deployments – if you launch it from Ubuntu WSL, it automatically spawns the actual server in Windows (so it can interface with Resolve's DLL), handling stdout/stderr across the boundary [3] [4]. This design proved effective: users can drive Resolve with natural-language prompts (e.g. "add a marker at 01:00") and MCP translates that to the proper API calls or key presses. The project's approach highlights how a mix of strategies – API for data queries and structured tasks, synthetic input for GUI-only interactions – achieves near-complete "human-like" control of the application.

- **REST API Bridge (davinci-rest):** As mentioned, this project takes a different route by *running inside* Resolve. It uses Resolve's built-in scripting (via the Fusion Scripting API) to start a FastAPI web server. Once running, any client (including a WSL script or remote machine) can hit endpoints like `/project/create` or `/timeline/add_clip` to drive editing operations [8] [23]. The provided Python client makes it easy to integrate into other pipelines. For example, an automated pipeline could detect new media files and then send an HTTP request from WSL to the Resolve REST server to import those files and place them on a timeline. This is essentially turning Resolve into a headless service for certain tasks. The limitation is you still need Resolve's GUI up and running (the server works inside it), and the available endpoints are only as comprehensive as the author implemented. It's an emerging project (few endpoints so far), but it demonstrates the possibility of a "headless API mode" for Resolve via a custom script.

- **Community Macro Scripts:** Outside of these frameworks, numerous editors and developers have rolled their own scripts for specific automation. Some use **Python + pywinauto** or **PyAutoGUI** to control Resolve, though (as noted) querying UI state is hard – often these scripts end up just sending hotkeys. Others have integrated **Stream Decks or MIDI controllers** by writing scripts that listen for device input (which can be done in WSL or Windows) and then perform sequences of Resolve actions. A real-world example: using a Stream Deck XL, each button could trigger a small script that either calls Resolve's API (via a python binding) or sends a series of keyboard shortcuts to perform an editing macro [24]. In these setups, WSL could run the listener and use something like `xdotool` –

but recall that *WSL GUI apps can't directly inject input into Windows apps* [25]. Instead, the script must call a Windows-side helper (like an AHK script or the SendKeys method). In summary, the community has explored many combinations: from simple one-off `.bat` or PowerShell scripts, to complex AHK frameworks (one AHK enthusiast even built a "Resolve Helper" script ~2,500 lines long to augment Resolve with custom shortcuts and behaviors).

- **Adjacent Tools:** The pattern of "logic on one side, UI control on the other" is not unique to Resolve. We see similar solutions for Adobe apps, for example. There's an *After Effects MCP* project under development that mimics Resolve MCP's approach for Adobe After Effects, allowing external agents to drive After Effects via its scripting interface (or via GUI) in a coordinated way [26] [27]. Blender, which has a Python API, has been controlled through chat-based prompts by running an internal server as well [28]. These parallels suggest a generalizable approach: use whatever official API is available for granular control, and supplement with OS-level automation for GUI actions, all orchestrated by an external "brain". In a production video pipeline, one might have WSL running scheduling and logic (perhaps pulling data from a database or receiving webhooks), and for any editing step that requires human software, it triggers Resolve (or Adobe, etc.) to do the work via these automation bridges.

Common Challenges and Gotchas

Automating a GUI application like Resolve through WSL involves several pitfalls to plan for:

- **Window Focus & Timing:** Sending keystrokes or clicks requires the Resolve window to be active and ready. If the window isn't focused, input might go nowhere or to the wrong app. Scripts therefore explicitly activate the Resolve window (`AppActivate` or `SetForegroundWindow`) and often insert a short delay before sending keys [11] [12]. Even then, race conditions can occur – e.g. if Resolve is busy (say, still loading a project), it might ignore keystrokes or process them late. It's wise to build in checks or waits. A common technique is to wait for a specific UI element or title change that signals Resolve is ready for the next step. For example, after launching Resolve or opening a project, an automation script might poll the Resolve API (if available) to see if the timeline is present, or use image detection to ensure the UI has loaded.

- **Focus Stealing:** Windows might pop up notifications or other apps can steal focus at the wrong time. If that happens in between your automation steps, keystrokes intended for Resolve could end up elsewhere. This is not just theoretical – users have reported scenarios where an automation unintentionally sent a "Ctrl+Q" to the wrong window because an update dialog appeared. When using SendKeys or AHK, always consider adding safeguards. For instance, check that the active window title *is* DaVinci Resolve right before sending a sensitive command (and refocus if not). As a cautionary note, Microsoft's documentation warns that **SendKeys** will send keystrokes to whatever has focus – if a different window suddenly activates, your input goes there [29]. Running your automation on a dedicated machine or virtual desktop (with notifications off) can mitigate random interruptions.

- **UI State Synchronization:** Many automation failures come from assuming an action is complete when it isn't. For example, triggering "Export" via keyboard shortcut and immediately trying to type a filename might fail if the export dialog hasn't appeared yet. GUI scripting often needs explicit waits or loops until the expected state is reached (e.g. wait until a window with title "Render Settings"

exists). With Resolve's API, some calls are synchronous (they return when done) but others might be asynchronous or have side effects that take time (for instance, adding many clips to a timeline might not reflect immediately). Testing your automation with different system speeds and project sizes is important to catch timing issues. Using **logs** is helpful – e.g., MCP logs to a file every action and any error messages from the API, which is invaluable for debugging timing problems.

- **Error Handling and Recovery:** If a step fails (maybe a shortcut didn't register, or a script API call returned an error), the automation should catch that and handle it. A gotcha is that once a sequence gets off track (say a menu didn't open when expected), subsequent inputs might wreak havoc (closing the wrong window, etc.). Strategies here include verifying after each step (did the panel actually open? Is the playhead where expected? etc.) and if not, attempting a recovery or aborting safely. In some cases, the best recovery is to reset the context – e.g. pressing `Esc` a few times to close any open dialogs and bring Resolve to a known state (like the Edit page), then trying again or reporting failure.

- **Permission and Environment Issues:** Tools like AutoHotkey or PowerShell might need to run with administrator privileges for certain actions (for example, AHK's keyboard hook or UIAccess features). If your WSL-initiated processes run into permission problems (can't simulate input due to UIPI/window isolation), you may need to adjust settings (e.g. run the script as the same user who launched Resolve, and not as a background service with lower rights). Likewise, if running on a headless server via RDP, note that some GUI automation won't work if the desktop is not interactive (Windows GUI apps often won't render or respond properly if no user session is active). Ensuring an active user session (or using tools like **VirtualVNC** to create a virtual display session for a logged-in user) can be necessary for reliable automation.

In summary, controlling DaVinci Resolve from WSL is absolutely feasible and has been done in practice using a combination of scripting APIs, PowerShell/AutoHotkey hacks, and creative architectural choices. Real-world projects like Resolve MCP show that with careful engineering – handling focus, timing, and bridging the WSL/Windows gap – you can achieve nearly hands-free editing operations [11] [12]. Just be prepared to incorporate robust error-checking and perhaps some trial-and-error tuning to get the automation as smooth as a human operator. With the right setup, WSL-driven scripts can orchestrate Resolve to do things like batch editing, rendering, or even respond to high-level commands, all without manual intervention. The key is combining the strengths of each approach: use the official API whenever possible for direct control, and augment with keyboard/mouse automation for the rest – essentially **letting the machine operate Resolve the same way you would, but at machine speed**.

Sources:

- Blackmagic DaVinci Resolve Developer Documentation and community forums (automation API and macro discussions)
- *DaVinci Resolve MCP* – Open-source AI integration and control server [7] [3]
- *davinci-rest* – REST API bridge for Resolve (FastAPI server inside Resolve) [8] [6]
- AutoHotkey Community – examples of Resolve automation scripts and UI hooking [16] [17]
- Reverse Engineering Q&A – UI Automation challenges with Resolve's Qt interface [20] [18]
- Stack Overflow/Super User – techniques for sending keystrokes via PowerShell (.NET `SendKeys` and WScript) [11] [12], and caveats about SendKeys timing/focus [13] [29].

---

[1] [2] [3] [4] [11] [12] [14] [15] [21] [22] 20260101_184119-src.txt
file://file_000000008454720690f99514933e508c

[5] [6] [8] [9] [23] GitHub - dev-beluck/davinci-rest: A REST API for DaVinci Resolve
https://github.com/dev-beluck/davinci-rest

[7] GitHub - samuelgursky/davinci-resolve-mcp: MCP server integration for DaVinci Resolve
https://github.com/samuelgursky/davinci-resolve-mcp

[10] View topic - pydavinci: a better way to script and automate Resolve
https://forum.blackmagicdesign.com/viewtopic.php?f=21&t=159133

[13] [29] telnet - SendKeys Method in Powershell - Super User
https://superuser.com/questions/1249976/sendkeys-method-in-powershell

[16] Automation Friendly Interface - Blackmagic Forum • View topic
https://forum.blackmagicdesign.com/viewtopic.php?f=33&t=129450

[17] Behringer X Touch One with DaVinci Resolve - 3 - AutoHotKey
https://www.youtube.com/watch?v=h0IZiYEkvLg

[18] [19] [20] python - C++, Windows UI Automation and DaVinci Resolve - Reverse Engineering Stack Exchange
https://reverseengineering.stackexchange.com/questions/30053/c-windows-ui-automation-and-davinci-resolve

[24] Looking for guidance on Resolve API capabilities and updates
https://www.reddit.com/r/davinciresolve/comments/l12s91/looking_for_guidance_on_resolve_api_capabilities/

[25] virtual machine - Can mouse clicks and key presses be sent to a Hyper-V VM from WSL2? - Super User
https://superuser.com/questions/1733040/can-mouse-clicks-and-key-presses-be-sent-to-a-hyper-v-vm-from-wsl2

[26] [27] [28] Claude AI + DaVinci Resolve / After Effects MCP : r/ClaudeAI
https://www.reddit.com/r/ClaudeAI/comments/1jc7cxy/claude_ai_davinci_resolve_after_effects_mcp/