ChatGPT

In-Call Device Switching – Development Specification

1. Header

Feature Name: In-Call Device Switching
User Story: *"As a user, I want to change my microphone and camera settings during a call so that I don't have to leave the meeting to fix technical issues."*
Feature Overview: This feature enables users to seamlessly switch their active microphone or camera **during an ongoing Zoom meeting**, without leaving or interrupting the call. Zoom already allows selecting different audio/video devices on the fly via in-meeting menus [1], and this specification aligns that capability with the Zoom Client's existing architecture. The implementation reuses Zoom's current audio and video infrastructure (services, managers, UI components) and introduces minimal new components needed to handle device switching logic. The goal is to integrate microphone/camera device switching into the call flow while preserving mute states, video on/off states, and leveraging existing mute verification and notification mechanisms. All new classes and interfaces follow Zoom's naming conventions (e.g. `ZoomClient.Audio.DeviceSwitchController_DSC001`) and fit into the established module structure.

Assumptions & Constraints:
- The user can access a UI menu (via the arrow next to the Mute or Stop Video buttons) to choose among **available microphone or camera devices** during a meeting [2].
- Switching devices **does not alter** the user's mute status or video-on/off status – if the user was muted or video off, they remain so after switching (and vice versa).
- The solution must **reuse existing services** (e.g. the Audio Service and capture pipeline) rather than creating new low-level media pipelines. New components are introduced only as orchestrators or to mirror an audio/video structure if one was missing.
- The design must maintain consistency with Zoom's software mute and hardware mute verification system: switching devices should not break the logic that verifies the microphone's muted/unmuted state (e.g., the existing audio state monitors should continue to function with the new device).
- All changes will be made in the Zoom Client (desktop application) codebase, and no server-side changes are required (the meeting back-end sees the same media streams, just from a different device source).

Version History:

Version	Date	Author	Description
1.0	Sep 24, 2025	Zoom Dev Team	Revised spec – aligned with existing architecture, reused components per Microphone Mute Verification feature

2. Architecture Diagram

System Components & Interactions: The high-level architecture is updated to include the new device switching logic, integrating with Zoom's existing client modules. The **ZoomClient** application (ZC001) comprises several modules: Audio Service, Video Service, Audio Verification, UI, etc. The new **DeviceSwitchController (DSC001)** resides in the Audio module and coordinates device changes. Below is an overview of how the components interact for in-call device switching:

- **User Interface (UI Module – UI001):** The in-meeting UI provides an arrow menu next to the **Mute button (UI002)** and **Video button** that lists available microphone or camera devices [2]. When the user selects a new device from these menus (e.g. chooses a different microphone or camera), the UI invokes the **DeviceSwitchController_DSC001** with the selection.
- **DeviceSwitchController_DSC001 (New, Audio Module):** This controller is the core of the feature. It receives the device selection event from the UI and determines whether it's an audio device or video device. It then calls into the appropriate service:
- For **microphone changes**: calls **AudioService_AS001** (Zoom Client's existing audio manager service) to switch the active input device. The AudioService in turn uses the **AudioCaptureService_AS002** to stop the current microphone stream and start the new one on the selected device.
- For **camera changes**: calls the new **VideoService_VS001** (a parallel video manager service) to switch the active camera. The VideoService uses **VideoCaptureService_VS002** to handle camera feed switching (turning off the old camera and initializing the new one).
- **Audio Service (AS001):** Existing module responsible for audio input/output management. It exposes an interface to select the active microphone device (e.g., `setActiveMicrophone(deviceId)`), which controls the **AudioCaptureService_AS002**. The DeviceSwitchController reuses this interface – no new low-level audio pipeline is created. The AudioService updates its internal state to reference the new microphone and delegates to AudioCaptureService to handle the stream transition.
- **AudioCaptureService_AS002:** Existing component that manages the audio stream capture from the microphone (start/stop, audio data pipeline). When instructed by AudioService to switch devices, it closes the current device handle and opens the new device (using OS audio APIs) for capturing. This service continues to feed audio data into the meeting and to any monitors, regardless of device changes. The switching occurs quickly to minimize audio disruption.
- **Video Service (VS001 – New):** New module analogous to AudioService, managing the camera device selection. It provides `setActiveCamera(deviceId)` to switch the video input. Internally, it commands **VideoCaptureService_VS002** to handle the camera stream. (If the Zoom client already had a video service, we extend it with device switching capability; if not, we introduce this as a new component consistent with AS001 structure.)
- **VideoCaptureService_VS002 (New):** Manages the video capture from the camera device. It starts/stops the webcam feed. On a camera switch, VideoService calls this service to close the old camera (releasing the device/driver) and initialize the new camera device. It ensures the video stream (if currently broadcasting) seamlessly continues from the new source, or simply updates the selected device to use when the user enables video.
- **Audio Verification Service (AVS001):** Existing module responsible for verifying and monitoring audio state (mute status, input activity). It includes:
- **AudioStateVerificationManager_AVS002:** Tracks the audio input state to ensure consistency (e.g., detecting if sound is coming through when unmuted, and silence when muted). It subscribes to audio data from AudioCaptureService_AS002. After a device switch, this manager continues to receive audio data from the new device via the same AudioCaptureService interface, so it remains

unaware of the change except for possibly reinitializing its reference to the device handle (if needed). **Integration:** The verification manager might momentarily detect a brief drop in audio during switching, but since the controller switches devices quickly, any gap is minimal. If the user is unmuted, audio levels from the new mic should be observed; if muted, the manager should continue seeing silence. No changes to AVS002 logic are required for this feature.

- **SoftwareMuteMonitor_AVS004:** Monitors the software mute state and ensures that when the user is muted in the app, no audio is actually sent (and conversely, that audio is being transmitted when unmuted) [1]. This component works with AudioCaptureService to detect any anomalies (it was used in the "Microphone Mute Verification" feature). **Integration:** When a device switch occurs, AVS004 will get updates from AudioService/AudioCaptureService about mute status and audio signal. It will continue to enforce that if software mute is on, the new device's audio is suppressed. It also helps detect "mute conflicts" – e.g., if the user is unmuted in software but the new microphone has a hardware mute on (resulting in no audio input). In such a case, AVS004 (in conjunction with any hardware monitor like AVS003 if present) will flag that no sound is coming despite unmute, and trigger the existing UI notification (the "microphone is muted" banner) to alert the user of a possible hardware mute or issue.
- **UI Notification Component:** The existing **NotificationBannerComponent_UI003** (from the mute verification feature) is repurposed to display any important alerts during device switching. For example, if a device switch fails or if a new device is detected but not functioning, this banner can show an error message (e.g., "Cannot access microphone. Please check your device."). On successful switches, a small confirmation might be shown (e.g., a brief toast like "Microphone switched to XYZ") – this could also be done via a similar UI mechanism. The goal is to integrate with Zoom's existing notification system rather than introducing new UI elements for alerts.

Overall Interaction: When the user selects a new device in the UI, the **DeviceSwitchController_DSC001** orchestrates the switch by invoking existing services for audio/video. The architecture remains layered – the UI layer talks to the controller in the Audio module, which in turn talks to the Audio/Video services, which interface with low-level OS APIs (e.g., audio drivers, camera drivers) to effect the change. The **existing audio stream pipeline is reused** (AudioCaptureService continues to be the source of audio frames, now from a new device) and the **existing video pipeline** is similarly used for new camera frames. The diagram below illustrates the component layout within the Zoom client and the new/updated interactions for device switching:

```
ZoomClient (ZC001)
├─ Audio Service (AS001)
│  ├─ AudioCaptureService_AS002
│  └─ DeviceSwitchController_DSC001 [new]
├─ Audio Verification Service (AVS001)
│  ├─ AudioStateVerificationManager_AVS002
│  └─ SoftwareMuteMonitor_AVS004
├─ Video Service (VS001) [new]
│  └─ VideoCaptureService_VS002 [new]
├─ UI Module (UI001)
│  ├─ MuteButtonComponent_UI002
│  ├─ VideoButtonComponent (existing)
│  └─ DeviceSelectionMenu_UI004 [new]
├─ Settings Service (SS001)
```

```
│   └─ UserSettingsManager_SS002
└─ Network Module (WS001)
    └─ WebSocketManager_WS002
```

Figure 1: Zoom Client architecture with In-Call Device Switching. The new **DeviceSwitchController_DSC001** (in Audio Service) connects the UI's device selection menu to the Audio/Video services. **Blue arrows** (in the conceptual diagram) indicate the flow of device switch commands: the UI menu triggers the controller, which calls AudioService_AS001 or VideoService_VS001 to switch devices. The AudioCaptureService_AS002 and new VideoCaptureService_VS002 interface with OS drivers (e.g., **AudioDeviceHandle_EXT001** and **CameraDeviceHandle_EXT003** respectively) to capture media from the selected hardware. Existing **Audio Verification (AVS002, AVS004)** components monitor the microphone's state and audio levels, working unchanged with the new device. The **MuteButton (UI002)** and **VideoButton UI** trigger their respective menus (UI004) and also handle mute/unmute or video on/off toggling (dotted lines indicate mute/video toggle flows, which remain as in the original design). The **NotificationBanner UI003** is used to alert the user of any issues (e.g., device errors or mute conflicts) during the switch. The **UserSettingsManager_SS002** may be updated with the user's new preferred devices. The **WebSocketManager_WS002** (network layer) continues to send audio/video streams to the meeting; it is not directly aware of device switching, aside from momentary pauses in the media stream during device reinitialization.

(Note: The architecture leverages the existing ZoomClient base classes – e.g., `BaseService_BS001` *for services like AS001, VS001 and* `BaseComponent_BC001` *for UI components – ensuring consistency with the overall client structure. The numbering of modules/classes follows the Zoom convention, with new classes given the next available IDs in their category.)*

3. Class Diagram

The class diagram below provides a static view of the key classes involved in the In-Call Device Switching feature and their relationships. It highlights new classes introduced (marked **[New]**) and how they associate with existing ones within the Zoom Client.

- **ZoomClient (ZC001)** – *[Main application container]*
- **AudioService_AS001** – *Manages audio devices and streams (part of ZoomClient.Audio).*
  - **AudioCaptureService_AS002** – *Handles microphone audio capture.*
  - **DeviceSwitchController_DSC001** – *[New]* Orchestrates switching of input devices (mic/camera).
- **AudioVerificationService_AVS001** – *Monitors and verifies audio state.*
  - **AudioStateVerificationManager_AVS002** – *Tracks audio stream state (used for mute verification).*
  - **SoftwareMuteMonitor_AVS004** – *Ensures muted state corresponds to no audio (and detects conflicts).*
- **VideoService_VS001** – *[New]* Manages video (camera) devices and streams (ZoomClient.Video).
  - **VideoCaptureService_VS002** – *[New]* Handles camera video capture.
- **UIModule_UI001** – *Contains UI components for in-call controls.*
  - **MuteButtonComponent_UI002** – *UI for mute/unmute (with device menu arrow).*

    - **VideoButtonComponent** – *UI for start/stop video (with device menu arrow) – assumed existing (similarly numbered in sequence).*
    - **DeviceSelectionMenu_UI004** – *[New]* Dropdown UI listing available mic/camera devices (invoked from UI002/VideoButton).*
    - **NotificationBannerComponent_UI003** – *UI for in-call notifications/alerts (used for errors or mute conflict messages).*
- **SettingsService_SS001** – *Manages user preferences/settings.*
    - **UserSettingsManager_SS002** – *Stores user's default device preferences, etc.*
- **NetworkModule_WS001** – *Handles network connection for meeting streams.*
    - **WebSocketManager_WS002** – *Sends/receives media data to/from meeting server.*

Relationships and Notable Associations:

- The **DeviceSwitchController_DSC001** is *invoked by UI components* (DeviceSelectionMenu in the UI Module) whenever the user chooses a device. It in turn *calls methods on* **AudioService_AS001** or **VideoService_VS001**. Thus, DSC001 acts as a controller/mediator between UI and the lower-level services.

- **AudioService_AS001** aggregates the **AudioCaptureService_AS002** (composition relationship) – it controls the lifecycle of AudioCaptureService. Similarly, **VideoService_VS001** aggregates **VideoCaptureService_VS002**. Both service classes provide interfaces to select devices and manage streaming.

- **AudioCaptureService_AS002** uses an external dependency **AudioDeviceHandle_EXT001** (not shown in detail) representing the OS-specific audio input device handle/ID. On switch, this handle is updated to the new device. Likewise, **VideoCaptureService_VS002** will use a **CameraDeviceHandle_EXT003** to interface with the OS camera driver. These external handles are managed through platform APIs but abstracted in the Zoom client.

- **AudioStateVerificationManager_AVS002** and **SoftwareMuteMonitor_AVS004** have observer relationships with the AudioCaptureService (they listen for audio level changes and mute toggle events). They do not directly interact with DeviceSwitchController; instead, they continue their normal operation on whatever device AudioCaptureService is currently capturing from. If the microphone device changes, AudioCaptureService will simply feed data from the new source – the verification components remain connected to the service (association via event callbacks).

- **MuteButtonComponent_UI002** and the Video button UI trigger the **DeviceSelectionMenu_UI004** (e.g., via a click on the arrow, probably handled inside UI002's code-behind). The menu UI then calls the DeviceSwitchController with the user's choice. The **UI components** call into the controller (association from UI to controller). The MuteButton and VideoButton also call AudioService_AS001/VideoService_VS001 for toggling mute or video on/off (these are existing interactions not changed by this feature).

- **NotificationBanner_UI003** is invoked by **SoftwareMuteMonitor_AVS004** or by the DeviceSwitchController/AudioService in case of errors. For instance, if a new device is not producing audio and the user is unmuted, AVS004 might trigger the banner saying "Your microphone appears to be muted or not working." In the case of a switching failure (e.g., device access denied), the DeviceSwitchController can directly use this component to show an error message.

- **UserSettingsManager_SS002** is used by **DeviceSwitchController** (or Audio/Video Service) to update the user's default device preferences when a switch is made. For example, if the user selects a new microphone, this preference can be saved so that next meetings default to that mic. The association here is that AudioService/VideoService likely queries and sets values in UserSettingsManager (so there's a link between those service classes and SS002).

(The class diagram is kept abstract; in implementation, classes like DeviceSwitchController may inherit from a base ZoomClient Service class (for consistency) and UI components inherit from BaseComponent. These details follow Zoom's standard patterns [3] but do not alter the relationships relevant to this feature.)

4. List of Classes

Below is a list of relevant classes/modules for this feature, with brief descriptions of their purpose and any new fields or methods introduced. New classes or significantly changed classes are marked **[New]**. We follow Zoom's naming convention by module prefix and unique ID.

- **DeviceSwitchController_DSC001** – **[New]** *Controller for in-call device switching.* Located in the ZoomClient.Audio module, this class coordinates the switching of microphone and camera devices during a call. It provides a unified interface (e.g., `switchDevice(deviceType, deviceId)`) that the UI calls when a user selects a new device. Internally, it calls `AudioService_AS001.selectMicrophone(deviceId)` for audio devices or `VideoService_VS001.selectCamera(deviceId)` for video devices. It handles logic such as checking the user's current mute/video state, invoking the appropriate service methods, and handling any exceptions/errors during the switch. This controller does **not** manage low-level media itself; it orchestrates existing services. It may also communicate with the UI (or NotificationBanner) to confirm success or report failures.

- **AudioService_AS001** – *Existing Zoom Client Audio Service.* Manages audio input/output and device selection. We reuse this service's capabilities to switch the microphone. In this feature, AudioService gets a new or existing method exposed to change the active input device (e.g., `selectMicrophone(deviceId)`). This method interacts with AudioCaptureService_AS002: it will stop the current mic stream and start the new one. AudioService keeps track of the current microphone device (e.g., a field `currentMicDeviceId`). **Reused Components:** AudioService already handles device enumeration and selection (e.g., listing devices for the settings UI); we leverage that rather than creating a new device manager. AudioService also notifies other components of device changes – for example, it can inform AudioStateVerificationManager that the source changed (if necessary).

- **AudioCaptureService_AS002** – *Existing Audio Capture Service (Audio Stream Control).* This service captures audio from the microphone and feeds it into the call (and to audio monitors). In the context of device switching, AS002 is extended to support dynamically switching the underlying device handle. It likely already supports opening a specified device at startup; we ensure it can also transition during runtime. New logic includes releasing the old audio device (closing the audio input stream) and initializing the new device (opening new stream) when instructed by AudioService_AS001. AudioCaptureService maintains the **AudioDeviceHandle_EXT001** for the microphone – this handle is updated to the new device's handle upon switch. Throughout the switch, AudioCaptureService ensures minimal interruption: if the user is unmuted, it stops capturing from Device A and immediately begins capturing from Device B, handing data to the meeting's pipeline; if the user is muted, it may still open the new device (or wait until unmuted) but not send any data.

- **AudioStateVerificationManager_AVS002** – *Existing Audio State Tracker.* Monitors the microphone's audio signal and mute state to verify that the software mute is effective (part of the mute verification

feature). It receives audio level info from AudioCaptureService_AS002 and mute toggles from AudioService_AS001. **Changes for this feature:** None to its core logic – but it inherently supports the feature by continuing to monitor the audio stream after a device switch. If the new microphone device is providing audio, AVS002 will see the audio levels as usual. If the user was muted or if the new device yields silence (e.g., hardware muted or faulty), AVS002 detects that and can work with AVS004 to alert the user. The class ensures *continuity*: any anomaly in expected audio (for instance, user is unmuted but no sound is detected after switching) is flagged.

- **SoftwareMuteMonitor_AVS004** – *Existing Mute Status Monitor.* Enforces and checks mute/unmute consistency, and raises user-facing alerts on issues. For this feature, AVS004 plays a role in maintaining the **software/hardware mute model consistency** during device switches. It listens for changes such as a new device being active and monitors if sound is coming through when it shouldn't or vice versa. **Example:** If after switching devices, the user is unmuted in software but due to hardware mute on the new mic no sound is transmitted, AVS004 (with help from any hardware monitor AVS003) identifies this mismatch (software says "unmuted" but effectively muted) and triggers the NotificationBanner_UI003 ("Your microphone is muted by hardware"). Similarly, if the user was muted in software, AVS004 ensures that remains true on the new device (it might double-check that no sound leaks during transition). No new fields are needed, but it subscribes to any "device changed" event from AudioService so it can log that or reset any needed state (like resetting any timers or counters used in verification).

- **MuteButtonComponent_UI002** – *Existing UI component (Meeting toolbar mute button).* Provides the user interface to mute/unmute and also contains the entry point for selecting audio devices. This feature does not modify the mute toggle functionality of UI002, but extends its behavior such that clicking the **dropdown arrow** on the button opens the **DeviceSelectionMenu_UI004** for microphones. (In the Zoom UI, this is the small arrow next to the mute icon [2].) The MuteButton component passes the list of available microphones (populated via AudioService) to the menu and handles user interaction. **Integration:** The UI component calls `DeviceSwitchController_DSC001.switchDevice("audio", deviceId)` when a microphone is chosen. It also likely updates its icon/state if needed (though switching device doesn't change the muted/unmuted icon). There are no new fields in UI002 except possibly a reference to the DeviceSwitchController or a callback hookup for device selection.

- **VideoButtonComponent (UI – existing)** – The meeting toolbar's Start/Stop Video button. Analogous to the MuteButton, it has an arrow menu for selecting the camera device [4]. We assume an existing component for this (if not previously specified, we introduce it conceptually). This component opens the device menu listing available cameras and calls the DeviceSwitchController (or directly VideoService) when the user picks a different camera. Like the mute button, it maintains the on/off state of video. When a camera is switched, if the video was on, the VideoButton might briefly indicate video is restarting (possibly showing a loading icon) and then continue showing the "on" state once the new feed is live. If video was off, switching simply changes the internal selection without altering the off state (the button stays "Video Off"). No new persistent fields; it uses existing video device list from VideoService.

- **DeviceSelectionMenu_UI004** – **[New]** *UI dropdown for device selection.* This is the new UI element introduced to list and allow selection of input devices during a call. It appears when the user clicks the arrow on the Mute or Video buttons. We implement it as part of the UI module, possibly as a

generic menu that can show both audio and video devices (or two separate instances for each context). It interacts with the AudioService_AS001 and VideoService_VS001 to get the lists of available devices (names and identifiers) to display. When the user clicks on a device in the menu, UI004 invokes the DeviceSwitchController (passing along whether it's a mic or camera and the selected device's ID). After selection, the menu closes and the Mute/Video button UI might update its label or icon (some UIs show the currently selected device name in the menu title). This component may be a simple extension of an existing menu class, but functionally it's introduced by this feature.

- **VideoService_VS001** – **[New]** *Video device service/manager.* This is a new service component (paralleling AudioService) that manages camera devices. If the Zoom client previously managed camera selection in a less formal way, we formalize it here. VideoService handles enumerating available cameras, selecting the active camera, and controlling the camera feed via VideoCaptureService_VS002. It likely inherits from the same base class as AudioService (for consistency in how services are structured [5]). Key responsibilities:

- Provide `selectCamera(deviceId)` method that the DeviceSwitchController or UI calls when switching the camera. This will stop the current camera (if active) and start the new one through VideoCaptureService.
- Store the current activeCameraId and maybe user preference for default camera (could interface with UserSettingsManager).
- Provide the list of cameras to the UI (similar to how AudioService provides mic/speaker lists).

- Ensure that if the user's video was on, switching cameras triggers the sending of new video stream to the meeting (and possibly a notification to other participants that the video is reconnecting if needed); if video was off, it simply prepares the new device for when video is turned on.

- **VideoCaptureService_VS002** – **[New]** *Video capture stream controller.* Handles the actual interfacing with the webcam hardware. Likely implemented using platform-specific camera APIs (DirectShow/MediaFoundation on Windows, AVFoundation on Mac, etc.), it opens the camera device and captures frames to send to the meeting's video pipeline. For this feature, its main function is to **switch camera input on demand**. Methods include starting/stopping video capture, and now a routine to switch device: e.g., `setCameraDevice(deviceId)` which internally will:

- If currently streaming and video is on, signal the old camera feed to stop (and possibly inform the UI to show a brief pause or freeze frame).
- Initialize the new camera device (open driver, start capture).
- Feed the video frames from the new camera to the encoding/sending pipeline (the meeting sees a brief transition but remains in the same video channel).
- If video was off, it may just store the new device as the default and not open it until the user hits "Start Video".
- Handle errors (device not found or cannot open) by throwing an error back to VideoService/Controller.

VideoCaptureService maintains a handle to the camera (like **CameraDeviceHandle_EXT003**) and likely uses existing Zoom video engine components for processing the video frames. It ensures continuity of the video stream to the extent possible (there will be an inevitable slight interruption when switching).

- **UserSettingsManager_SS002** – *Existing settings manager.* Stores user preferences such as default microphone, speaker, camera, etc., in the Zoom client's configuration. When a user switches devices in-call, we may update these preferences: for example, if the user chose a specific microphone, the application can set that as the new default for future meetings (unless the user has "Auto-select device" enabled). In this feature, **AudioService_AS001** and **VideoService_VS001** will interface with UserSettingsManager to read the initial preferred devices when a meeting starts and update the preferences when the user manually switches. Fields involved might include `preferredMicId`, `preferredSpeakerId`, `preferredCameraId`. This ensures the next meeting will use the last selected device (so the user doesn't have to switch every time). This class's schema is updated if needed to include camera device preference (if it wasn't already present).

- **NotificationBannerComponent_UI003** – *Existing in-call notification UI.* This UI element (already introduced by the mute verification feature) is used to display warnings or info to the user on top of the call (e.g., "You are muted" prompt when trying to speak). In the context of device switching, the same component is utilized to show:

- Error messages: e.g., "Camera could not be started. Try a different device." if a switch fails.
- Mute conflict alerts: e.g., if after switching, no audio is detected while unmuted, it might flash "Your microphone appears to be muted or not working" (which was the original use-case for this banner).
- Possibly confirmation messages: though not typically needed (the UI menu itself indicates the selected device), a quick confirmation could be shown especially if the user's action has an effect (some applications show a temporary label "Microphone: Yeti Nano" on switch). If implemented, it would be via this banner or a similar transient UI element.

No changes in implementation are needed; DeviceSwitchController or the services will call existing methods to display messages through this component.

- **WebSocketManager_WS002** – *Existing network transport for media.* It maintains the live connection to Zoom's meeting servers, sending encoded audio and video data. From a class perspective, it is unaffected by the device switching logic – it continues to send whatever audio/video data it receives from the capture services. However, it's worth noting that a device switch might cause a brief pause in media transmission (audio might drop for a second, video frame might freeze) while the new device is initialized. The WebSocket/meeting protocol should handle this seamlessly (it's akin to a momentary network glitch or silence – no new class behavior). There is no new interface here; the same audio/video streams are used.

(Other existing classes not listed remain unchanged. For instance, HardwareAudioMonitor_AVS003 – if part of the audio verification system – would update its monitoring to the new microphone's hardware mute state when switched, ensuring continuous hardware mute detection. We rely on that existing behavior to catch scenarios where the new device is hardware muted.)

5. State Diagrams

Device Switching State Machine (Microphone Example): The following state diagram illustrates the microphone's state transitions during a device switch, including the mute status. A similar logic applies to the camera (with "video on/off" replacing mute). This diagram emphasizes that switching devices does not alter the mute state, and shows the possible states before and after switching:

- **State A:** *Using Mic Device A – Unmuted (audio is being transmitted)*
- **State B:** *Using Mic Device A – Muted (no audio transmitted)*
- **State C:** *Using Mic Device B – Unmuted*
- **State D:** *Using Mic Device B – Muted*

Transitions:
- **Mute/Unmute toggling:** A ↔ B and C ↔ D. When the user clicks the mute button, the system toggles between transmitting audio and silence for the *current* device. This is existing behavior: it doesn't depend on device switching, but it's shown to illustrate continuity (the user can mute before or after a switch).
- **Device Switch (while unmuted):** A → C on "Switch to Device B" event. If the user is actively talking on Device A (unmuted) and chooses Device B, the system will transition to state C. **During this transition**, AudioCaptureService stops Device A and starts Device B. There may be a sub-state "Switching in progress" (brief) where no audio flows, but from the user's perspective, it's an atomic action – they move from A to C, staying unmuted. After switching, audio is now coming from Device B.
- **Device Switch (while muted):** B → D on "Switch to Device B". If the user is muted on Device A and switches, since no audio is being sent, the switch happens silently in the background. The state goes from B to D: the user remains muted, now with Device B selected. AudioCaptureService may not even start Device B's stream until unmuted (depending on implementation; some apps open it anyway but drop the data). The key is that from the user perspective, they are still muted after switching. When they eventually unmute (transition D → C), audio will come from Device B.

These state transitions ensure **the mute status is preserved** across device changes. The software mute flag doesn't change due to a switch action alone. If the user was unmuted, they remain unmuted and the new device immediately becomes live; if they were muted, they remain muted until they choose to unmute.

Additionally, we consider **hardware mute state**: Each device might have its own hardware mute or physical condition. This isn't explicitly shown in the software state diagram, but it's handled by the verification system: - For example, suppose we were in State A (Device A unmuted) and the user switches to Device B. If Device B happens to have a hardware mute on (or some issue), the system will attempt A → C, but effectively what happens is that after switching, no sound is captured. Software-wise, we are in "unmuted on B", but hardware is preventing audio. This results in a *mute conflict scenario*. The **SoftwareMuteMonitor_AVS004** will detect that "unmuted but silent" condition and likely flag an alert. In terms of states, one could consider that as a separate error state, but since the software still thinks it's unmuted, we rely on the notification to alert the user. The user can then realize the new mic is hardware-muted (and either press its button or switch back). The design ensures such a conflict is handled as it was prior to switching (the same mechanism that caught a hardware mute on Device A works for Device B) – thus the **state model (muted vs unmuted)** remains consistent, and conflicts are resolved via user notification rather than by automatically changing state.

For the **camera device switching**, the states are analogous: - *Video On with Camera A*, *Video Off with Camera A*, *Video On with Camera B*, *Video Off with Camera B*. - Switching camera while video is on will briefly pause the video (likely freezing the last frame from Camera A) and then resume with Camera B's feed – effectively *Video On A → Video On B* transition. - Switching while video is off simply changes the selection (no video feed active) *Video Off A → Video Off B*. The video-on state (off in this case) persists.

These state transitions confirm that the feature's behavior aligns with user expectations: changing devices does not by itself mute or unmute you, nor start or stop your video; it only changes *which* device is used when those streams are active [4].

6. Flow Chart

The following is a **flow of events for switching devices during a call**, showing both the user's actions and the system's responses. It covers both microphone and camera switching scenarios, highlighting the branching where appropriate:

1. **User initiates device switch:** The user is in an ongoing meeting and decides to change their input device (microphone or camera). They click the **dropdown arrow** next to the Mute button (to change microphone) or next to the Stop Video button (to change camera) [2] [4].
2. **UI displays device list:** The Zoom client UI (DeviceSelectionMenu_UI004) opens, showing a list of available devices of that type:
3. If it's the audio menu: it lists all connected microphones (and possibly audio outputs, though speaker switching is outside this user story).
4. If it's the video menu: it lists all available camera devices. (The UI obtains these lists via AudioService_AS001 and VideoService_VS001, which query the OS or cached device info.)
5. **User selects a device:** The user clicks on one of the listed devices (e.g., selects "External Webcam" instead of "Laptop Camera").
6. **UI calls controller:** The selection event triggers a call to `DeviceSwitchController_DSC001.switchDevice(deviceType, deviceId)` with the chosen device's identifier and type. For example, `switchDevice("audio", "MicDevice3")` or `switchDevice("video", "CameraDevice2")`. The UI immediately closes the menu.
7. **DeviceSwitchController processes request:**
8. The controller logs the request and determines the type:
    - **If deviceType == "audio":** It calls `AudioService_AS001.selectMicrophone(deviceId)`.
    - **If deviceType == "video":** It calls `VideoService_VS001.selectCamera(deviceId)`.
9. It also checks the current state (mute or video on/off):
    - For audio: it queries if the user's mic is currently muted via AudioService/AVS004.
    - For video: it checks if the video is currently on via VideoService/VideoButton UI state.
10. These states may influence how the services perform the switch (as described next).
11. **AudioService/VideoService perform switch:**
12. **Audio path (switch microphone):**
    a. AudioService compares the `deviceId` with the current one. If it's different (as expected), it proceeds.
    b. It instructs **AudioCaptureService_AS002** to switch to the new mic: - If the user is **unmuted** (AudioService knows the mic is active): 1. AudioCaptureService **stops** the audio stream from the old device (flushes/ends capture on Device A). 2. It then **initializes** the new microphone (Device B) – opens the audio input stream. 3. It **begins capturing** audio from Device B and feeding it to the

meeting. This might happen almost immediately; the AudioService might cross-fade or simply accept a brief gap in audio during the change. 4. AudioService updates its `currentMicDeviceId` to B and might emit an event “activeMicChanged” to notify other components.
- If the user is **muted**: 1. AudioCaptureService may either fully stop capturing from Device A (since it might not have been capturing at all due to mute, depending on implementation) or simply note the change. 2. It switches internal device references to Device B. It might open Device B in a standby mode (some apps open the device even if muted, but immediately discard the data; others wait until unmuted to actually start). 3. It does **not** start transmitting audio because mute is on. So from the meeting’s perspective, nothing changes (still no audio). 4. AudioService updates the current device id to B. 5. No immediate audio verification needed since muted, but it could check that Device B is at least accessible (maybe it could briefly capture to ensure it works, then mute it – an implementation detail not exposed to user). c. AudioService returns a status to DeviceSwitchController indicating success or failure.

13. **Video path (switch camera):**
    a. VideoService receives the `selectCamera` call. It likely has a similar flow: - It instructs **VideoCaptureService_VS002** to switch to the new camera device. - If **video is currently ON** (user’s camera was broadcasting): 1. VideoCaptureService stops the feed from Camera A (informs the video pipeline to pause or ends the video stream from A). 2. It initializes Camera B (this might involve a slight delay if the camera’s driver needs time). 3. It starts capturing from Camera B and feeding frames to the video pipeline. Zoom might show a spinner or freeze the last frame until new frames arrive. 4. VideoService updates the active camera id and possibly notifies the UI (so the UI could update any camera name label if shown). - If **video is OFF**: 1. The service likely does not turn on Camera B. It may optionally open it for a quick test or just set it as “selected”. 2. It updates the default selection to B. 3. The actual video feed remains off. (No user-visible change except that if the user opens the menu again, the checkmark would now be on Camera B.) b. VideoService returns success/failure to the controller.
14. **Post-switch update and notifications:**
15. If the switch is **successful**:
    - The DeviceSwitchController logs success.
    - The UI is updated: typically, the menu arrow icon might show the new device as selected (in many UIs, the currently selected device has a checkmark in the menu – that’s handled by the menu component itself when next opened). The user now speaks through the new mic or the video feed switches to the new camera.
    - A small on-screen confirmation may be displayed. Though not strictly necessary, some feedback can be helpful (for instance, a tooltip that briefly says “Microphone switched to <device name>”). This would be done via **NotificationBanner_UI003** or a similar ephemeral message. If implemented, the controller or service triggers it.
    - The AudioStateVerificationManager_AVS002 and SoftwareMuteMonitor_AVS004 automatically continue monitoring. AVS002 might reset any internal audio level history due to the new device, but quickly resumes checking audio levels on the new stream. If the user is unmuted, AVS002 will see if audio is coming in; if not (e.g., user hasn’t spoken yet or device is silent), AVS004 may be on standby to warn after a few seconds if absolutely no audio is detected while unmuted. If the user is muted, nothing changes in the verification logic.
    - The **UserSettingsManager_SS002** is updated with the new default device (persisting the choice). For example, `preferredMicDeviceId` is set to the new device’s ID (so next time, Zoom will auto-select that mic). Similarly for camera. This step is handled inside AudioService/VideoService after a successful switch.

16. If the switch **fails** (error case):
    - Possible failures include: the selected device is not available (e.g., just disconnected), the OS returns an error when opening it, or permission is denied (for instance, macOS might pop "Zoom wants to access the camera" – if the user says no, it fails).
    - DeviceSwitchController or the respective service catches this error.
    - The system will **retain the previous device** in use (if the old device was working, Zoom should continue using it rather than dropping audio/video entirely). AudioService/ VideoService in such a case would roll back to the last known good device. If the old device is gone (e.g., user unplugged it and then the attempt to switch to a new one failed), then the app is left with no input – an error state where user has no working mic/cam. This is rare, and the user would have to either plug something back in or resolve the permission.
    - The user is **notified of the failure** via an error banner (UI003). For example, "Unable to switch to <device name>. Device not found or inaccessible." If the OS provides a specific error (like "device in use by another application"), it could be shown.
    - Logging: The error is logged for diagnostic purposes.
    - The UI might reopen the device menu automatically or highlight that selection failed (perhaps keep the checkmark on the old device). Typically, Zoom would just revert to the old selection quietly and inform the user via message.
    - The controller returns an error state, but since this is initiated by UI, there isn't a calling function to return to – instead, the user just sees the message and knows the switch didn't happen. They remain on the old device in this scenario (if available).
17. **End state:** The flow concludes with the user now using the chosen device. The meeting continues without needing to be left or rejoined. All modules settle into normal operation with the new configuration:
18. Audio/video streams flow from the new devices through the same pipeline.
19. The user can continue to toggle mute or video as before, now affecting the new devices.
20. The device menus remain available, so the user could switch again if needed (the process can be repeated as often as desired). The system is designed to handle repeated switches (though rapidly switching devices is not common, the code should still handle back-to-back requests gracefully, perhaps by queueing or ignoring a new request until the current switch completes).

Alternate scenarios:
- *Auto device change due to device removal:* (Though not explicitly user-triggered, consider if a device disconnects mid-call.) If the microphone or camera in use is unplugged or turned off, Zoom could automatically fall back to another available device and possibly show a notification ("Device A disconnected, switched to Device B"). This use-case is adjacent to our feature. The **DeviceSwitchController** could be leveraged for this as well: e.g., an event from the OS/AudioService about device removal could call `switchDevice` to a default alternative. This is an extension where the controller helps maintain continuity even without user action. The spec's architecture supports this, though the user story focuses on manual switching.

Overall, this flow ensures that the user's action of selecting a new device is handled quickly and transparently, with minimal disruption to the call. The critical parts – stopping the old stream and starting the new one, updating the UI, preserving mute states – are all done within a second or two in practice. Users do not need to leave the meeting or dig into settings; the in-call menu and our integrated controller handle it in real-time.

7. Development Risks and Failures

Implementing in-call device switching entails certain technical risks and potential failure modes. Below, we outline these along with how the design mitigates them:

- **Risk: Device switch failure due to hardware or driver issues.** For example, the new device might be in use by another application, not plugged in properly, or lacking OS permission. This could cause the `AudioCaptureService` or `VideoCaptureService` to throw an error when opening the device.

- *Mitigation:* The system is designed to catch such errors and revert to the last known working device. We use try-catch around device initialization. On failure, an **error notification** is shown to the user (via UI003) so they know the switch didn't occur. The audio/video services will continue using the previous device if possible. This ensures the user isn't left without any audio/video. Logging the error will help troubleshoot specific causes (e.g., "Access denied to camera"). We also advise through UI for the user to check device connection or OS settings if this occurs.

- **Risk: Brief interruption of audio/video during switching.** Stopping one device and starting another can cause a gap in the media. Audio might drop out for a moment, or video might freeze while switching. If not handled smoothly, participants might miss a word or see a visual glitch.

- *Mitigation:* The switch process will be as fast as possible, and Zoom's codecs will treat it like a momentary glitch. On audio, the design accepts a very short silence as preferable to overlapping streams; on video, participants may see a frozen frame or spinning indicator for a second. This is similar to what happens if network blips – the system can recover. We ensure **state transitions are atomic**: the old device is only closed right when the new one is ready to start (especially for audio, we might overlap a tiny buffer from the old device if feasible to cover the gap). Additionally, if the new device initialization supports it, we could prepare the device in a "warm standby" if muted (so unmute is instant). In practice, a minor interruption is unavoidable but should be minor. The user can be warned (in documentation or training) that switching devices mid-sentence might cause others to not hear a word or two, so it's best done when not actively speaking.

- **Risk: No audio after switching (user unmuted but new mic is silent).** This could happen if the new microphone's hardware mute is on, its volume is very low, or it's malfunctioning. The user might not realize immediately and could continue talking without being heard.

- *Mitigation:* This scenario is exactly why Zoom has the **AudioStateVerificationManager_AVS002** and **SoftwareMuteMonitor_AVS004**. These will detect that the user is unmuted in software but the audio input is essentially zero. Within a second or two of the switch, if the user speaks and nothing is picked up, the system will trigger the existing "mute conflict" workflow – likely flashing a banner "You are muted" or "No one can hear you" (even though software mute is off) [1] . This prompts the user to check the new mic's hardware mute or connections. Essentially, the **existing mute verification feature is our safety net**: it remains in effect with the new device. The risk of prolonged silence is thereby reduced; the user gets notified quickly if their action didn't result in audible speech.

- **Risk: Inadvertent audio leakage or privacy issue during switch.** For instance, if the user was relying on a hardware mute on the old device and forgets that Zoom is unmuted, switching to a new

device that isn't hardware-muted could unexpectedly broadcast audio. Or switching cameras might accidentally turn on a camera feed the user wasn't ready to share (if they thought video was off but it was actually on).

- *Mitigation:* The design principle is **never change the user's mute or video-on state automatically**. So if they were effectively muted (even via hardware) before, they should remain effectively muted after. In the example: user had mic A hardware-muted and Zoom unmuted (so they were silent but in a conflict state), then switches to mic B. Mic B might be live (not hardware-muted), so suddenly their voice is transmitted. This is a user scenario that can happen; we rely on user awareness here since the software cannot know user's intention (Zoom showed them they were unmuted the whole time). However, our feature doesn't exacerbate it – it behaves consistently with Zoom's model (if you are unmuted in Zoom, any functional mic will transmit). We do mitigate by quickly alerting if needed (as above). For video, if a user's video was actually on (showing blank because maybe lens cap on) and they switch to another camera that's uncovered, they will start showing – but again, Zoom's UI always indicates if video is on with a red camera icon, etc. So as long as the user knows their state, there's no silent change. We ensure that **video off stays off** when switching cameras, preventing accidental video start. And if video was on, we assume the user consents to showing whatever the new camera sees.

- Additionally, **OS permission prompts** act as a safeguard: if Zoom had no permission for the new camera, the OS will prompt the user to allow it. If they deny, no video is shown (and we handle the failure as above). This gives the user control and awareness.

- **Risk: Rapid or concurrent switching actions.** A user might quickly switch devices multiple times (e.g., cycle through mics testing which works), or simultaneously try to switch mic and camera in quick succession. This could stress the system or cause race conditions (e.g., two switches overlapping).

- *Mitigation:* The DeviceSwitchController can serialize device switch requests. If a switch is already in progress, further requests could be queued or ignored until completion. We'll implement a simple lock: when one switch starts, the controller will not start another until it finishes (or times out). Given user interaction speeds, it's unlikely they legitimately try two at once, but this prevents edge-case bugs. Also, since audio and video switching are handled in the same controller, it ensures an atomic sequence (we wouldn't try to switch camera and mic at the exact same millisecond – one will happen then the other). This avoids weird interactions like audio switching half-done when video switching begins. Proper thread-safety in AudioService/VideoService (if they run on different threads) is also ensured by their internal design or by performing all switch operations on Zoom's main/UI thread or a dedicated media thread with synchronization.

- **Risk: Cross-platform differences and device ID management.** Zoom runs on multiple OS's – device enumeration and identification differ (e.g., Windows device IDs vs. macOS device unique IDs). Ensuring that the ID passed from UI through DeviceSwitchController to the services correctly identifies the same device in the native API is critical. If mismanaged, the wrong device might be activated.

- *Mitigation:* We rely on the existing AudioService and VideoService for device identification. These services likely maintain a mapping of friendly names to OS device IDs. The DeviceSelectionMenu

might pass a logical ID or index that the service knows. We will use the same mechanism Zoom uses in settings: for example, Zoom might label devices by name internally but use an underlying GUID. Our implementation doesn't change that – it just invokes the selection by that ID. We will thoroughly test on each platform to ensure the right device is engaged. This risk is mostly mitigated by reusing Zoom's proven device handling code rather than writing new enumeration code.

- **Risk: Integration with existing features and edge cases.** The introduction of mid-call switching touches various parts (UI, audio pipeline, settings). Potential edge cases:

- The **UI state** must remain consistent. For example, if the user opens the menu and the meeting ends or they quickly mute/unmute, the interactions shouldn't cause errors.
- If a user changes their default devices in the Settings dialog *during* a call (Zoom allows accessing settings mid-call), it could conflict with our in-call selection. We need to define which "wins" or how they sync. Likely, in-call selection updates the active device immediately, and settings UI will reflect that change if opened.

- *Mitigation:* We ensure that any global setting change also routes through the same AudioService/VideoService so that it uses DeviceSwitchController logic or vice versa. Essentially, there should be a single source of truth for active device at a time. Testing these interactions is necessary. We also ensure that stopping/starting video (by user action) near simultaneously with switching camera doesn't leave the camera on by mistake (the VideoService should handle a switch request by knowing if video is currently off or on as described).

- **Risk: Completion and QA risks.** (Project management perspective) Introducing changes to critical paths like audio/video could delay release if not tested well.

- *Mitigation:* We will carry out extensive testing for all combinations: switching audio while muted/unmuted, switching back and forth, switching right after joining a meeting, switching when only one device is available (should hide the menu arrow if no alternative device), etc. Testing on Windows/macOS/Linux with various devices (USB, built-in, Bluetooth) to ensure broad coverage. We'll also include automated tests for the controller logic (where possible, simulate the calls and see state outcomes) and manual tests for actual device switching.

Overall, while there are several risks, our design either **reuses existing robust components** (reducing new bug surface) or includes specific checks and user feedback loops to handle them. The critical failure scenarios (no audio or no video) are mitigated by Zoom's verification and notification system, ensuring the user is quickly informed and can take corrective action.

8. Technology Stack

This feature will be implemented within the existing Zoom client technology stack, utilizing the same languages, frameworks, and APIs that the Zoom application already uses for audio/video handling and UI. Key elements of the tech stack include:

- **Programming Language:** The Zoom desktop client core is primarily written in **C++** (with some modules possibly in Objective-C++ on macOS, etc.). We will implement the DeviceSwitchController and any service changes in C++ following the project's coding standards. The use of C++ allows direct

interaction with low-level OS APIs for device control and is consistent with Zoom's performance requirements for real-time media.

- **User Interface Framework:** Zoom's UI on desktop is built with a native UI toolkit (likely **Qt** or a custom C++ UI framework, as indicated by the cross-platform nature). We will create the DeviceSelectionMenu UI component using this same framework – e.g., in Qt, that might be a QMenu or QWidget that lists devices. The UI components (UI002, UI004, etc.) are integrated in the existing UI hierarchy. We ensure the UI follows the established patterns (for example, using Qt's signals/slots or message passing to communicate the selection event to C++ controller code).

- **Audio/Video Device APIs:** We leverage **operating system multimedia APIs** through Zoom's existing abstractions:

- On **Windows**, the audio capture uses WASAPI or DirectSound via Zoom's audio module; camera capture uses Media Foundation or DirectShow. We do not call these directly from our feature; instead, we call Zoom's AudioCaptureService and VideoCaptureService, which internally use these APIs. However, for understanding: switching devices essentially involves calling `IMMDeviceEnumerator` (on Windows) to get the device by ID and then initializing an `IAudioClient` for audio, or similarly selecting an IMFActivate for video capture devices. The heavy lifting is already done by Zoom's engine – our code just passes the correct device identifiers to those interfaces.
- On **macOS**, core audio (AudioQueue or AVAudioSession) is used for mic, and AVFoundation (AVCaptureSession) for camera. Again, the Zoom client has wrappers; we ensure our calls supply the right device UID to those wrappers.

- On **Linux**, typically ALSA/PulseAudio for sound and V4L2 for video. The approach is analogous.

- **Zoom Media Engine:** The client likely has an internal media engine (which could be partly based on libraries like WebRTC or custom). Regardless, the engine handles encoding/decoding, network send/receive. Device switching plugs into the input side of this engine. We aren't modifying the engine – just feeding it from a different source mid-call. The engine is capable of handling dynamic source changes (it might re-initialize the encoder for video if resolution changes with a new camera, etc.). Using the existing engine ensures continuity of media transport (no need to re-negotiate with the server, as the stream (RTP) continues in the same session, just with maybe a keyframe for video after switch).

- **Threading and Concurrency:** Real-time audio/video handling often uses dedicated threads (e.g., one thread continuously reads mic audio buffers). Our controller runs likely on the UI thread when invoked, then instructs services which might perform device init on a worker thread or same thread depending on design. We will synchronize via locks or message passing as provided by Zoom's infrastructure. For example, Zoom might have a task queue for audio operations to avoid blocking the UI – our `selectMicrophone` might post a task to that queue. The tech stack includes whatever concurrency primitives or event loop Zoom already uses (which could be Qt's event loop or custom).

- **Data Storage:** The user preference updates (for default devices) are stored in Zoom's configuration (possibly a JSON or plist or registry entries). We will use the existing **SettingsService** mechanism to save these. No new database or file format is introduced; we simply add/update fields in the config

schema if needed. The technology for that is likely standard file I/O through Zoom's config manager (which might be C++ code writing to disk, or OS-specific storage).

- **Testing Tools:** Not exactly stack, but we'll use Zoom's internal logging system (for debug logs when a device switch happens or fails). Also, internal testing may use virtual audio devices or camera simulators; if needed, we could employ tools like **virtual webcam software** or scripts to simulate device changes to test reliability.

- **No External Libraries added:** We intentionally do not add new third-party dependencies. Everything needed (UI components, device enumeration) is available within existing libraries Zoom uses (Qt, OS SDKs). Keeping within the current stack ensures compatibility and reduces footprint.

- **Build & Deployment:** The code will be integrated into the Zoom client application and built with the existing build system (likely CMake or similar, producing the executables for each platform). The feature will be behind the scenes for users with no extra installation. Possibly feature flags can toggle it if needed for gradual rollout, controlled via config (Zoom might have a mechanism to disable new features if issues arise). The stack remains the same in production – just an updated app binary.

Overall, the technology stack for this feature is essentially "Zoom's native client stack (C++/Qt on desktop, OS multimedia APIs)". By sticking to known technologies and Zoom's frameworks, we ensure maintainability and performance. The feature operates at the UI and application logic level, orchestrating things, rather than introducing any new low-level tech.

9. APIs

This section outlines the key APIs and interfaces (internal to the application and external OS interfaces) that the In-Call Device Switching feature will use or provide. We focus on new or modified APIs in the Zoom client as well as how we interact with OS-level APIs for device control.

Internal Zoom Client APIs / Interfaces:

- **AudioService_AS001**:
- *New Method*: `bool selectMicrophone(std::string deviceId);` – This API is exposed by AudioService to allow switching the active microphone. **Usage:** Called by DeviceSwitchController when an audio device is selected. **Behavior:** It stops the current audio capture (if any) and initializes the new device via AudioCaptureService. Returns `true` on success or `false`/throws on failure.
- *Existing Methods Used*: `getAvailableMicrophones()` – returns list of microphone devices (id, name) for populating the UI menu. We invoke this when building the device list UI. Also, `getCurrentMicId()` might be used to show which device is currently active (for marking in the UI menu).

- *Event/Callback*: AudioService may emit an event `onMicrophoneChanged(deviceId newDevice)` to notify other components (e.g., so UI or settings knows the current device changed). The DeviceSwitchController will subscribe or be implicitly aware via the call return. We ensure that AudioStateVerificationManager is also listening if needed (though it could also just detect via audio stream changes).

- **AudioCaptureService_AS002**:

- *Modified/Internal API*: We utilize an internal function such as `openDevice(deviceId)` and `closeDevice()` within AudioCaptureService. The `selectMicrophone` in AudioService will effectively call these in sequence. These were likely existing (for initial device open, and for cleanup). We ensure they can be invoked not only at startup but also at runtime for switching.

- *No direct public API to UI*, but internally it implements the necessary OS calls (below). We might add a private method like `switchDevice(deviceId)` as a wrapper that calls `closeDevice(); openDevice(newId);`. The design maintains encapsulation: AudioCaptureService abstracts OS details behind these methods.

- **VideoService_VS001**:

- *New Method*: `bool selectCamera(std::string deviceId);` – Similar to selectMicrophone, to switch the active camera. Called by DeviceSwitchController when user picks a new camera. It interacts with VideoCaptureService.
- *Existing/Related Methods*: `getAvailableCameras()` – list available cameras (for UI). `getCurrentCameraId()`. Possibly an existing method if Zoom had any camera selection (maybe not – if not, we implement it).

- *Event/Callback*: `onCameraChanged(deviceId)` event could be emitted to notify UI (for example, if there's a preview window in Zoom's UI showing your own video, it might want to reattach to the new camera feed – but that is probably handled automatically by video pipeline).

- **VideoCaptureService_VS002**:

- *Internal Methods*: `startCapture(deviceId)` and `stopCapture()`. We will use these for switching. Implementation-wise, `selectCamera` in VideoService will likely call something like:

```
if(cameraCapture.isRunning()) cameraCapture.stopCapture();
bool ok = cameraCapture.startCapture(newDeviceId);
```

  If `ok` is false, VideoService returns failure. If video was off, we might not startCapture immediately; instead we might just set a field `pendingDeviceId` to use next time video is started (or we start and immediately pause it). These decisions are encapsulated.

- VideoCaptureService might already exist if Zoom allowed camera selection in settings; we are making sure it supports runtime switch. Possibly adding a method `switchToDevice(deviceId)` that handles the logic in one call (stop old, start new if needed).

- **DeviceSwitchController_DSC001**:

- *New Interface*: This is a high-level interface between UI and services. It might not be a traditional "API" since it's an internal controller, but we can define:

    - `void switchDevice(DeviceType type, std::string deviceId);` – The main entry point. DeviceType could be an enum {Audio, Video}. This function contains the logic described earlier: route to the correct service's select method, handle errors, coordinate with UI. Because it doesn't return a value to UI (the UI call likely doesn't expect a synchronous return beyond maybe success), it handles success/failure internally (maybe updating UI or calling NotificationBanner on failure). If we do want to acknowledge success, we could have it return bool or use a callback to UI. Typically, though, the UI menu doesn't need confirmation – if something goes wrong, the error banner suffices.
    - This controller might also have helper methods or even overloads if needed, but likely one method is enough. It might also expose `switchAudioDevice(id)` and `switchVideoDevice(id)` internally that call the common routine.

- **MuteButtonComponent_UI002 / VideoButton UI**:

- *UI event integration*: These UI components use existing UI event APIs. For instance, in Qt, clicking the arrow might trigger a slot or signal. We integrate by connecting that UI event to a new handler that calls `DeviceSwitchController.switchDevice`. In pseudo-code:

```
void onAudioDeviceMenuAction(QString deviceId) {
    DeviceSwitchController.switchDevice(AUDIO, deviceId.toStdString());
}
```

  The UI menus likely use an action-per-device pattern; we'll wire those actions. No new public API, just hooking into the UI framework's event system.

- **NotificationBanner_UI003**:

- Already has an API like `showMessage(string text, MessageType type, duration)` which we use. For example, `NotificationBanner.showMessage("Microphone switched to Yeti Mic", INFO, 3s);`. Or `MessageType::Warning` for errors. We just call this at appropriate times. This is an internal UI API (the Banner component exposes it to controllers).

- **UserSettingsManager_SS002**:

- Provides APIs `setDefaultMic(deviceId)`, `setDefaultCam(deviceId)` or similar, and corresponding getters. We call these after a switch to save the choice. These might already exist if Zoom's Settings dialog uses them. If not, we introduce these calls. It likely interfaces with a config file save under the hood.

- **HardwareAudioMonitor_AVS003 (if applicable)**:

- Possibly an internal API to update which device's hardware mute state to monitor. If a new mic has a hardware mute (like a physical mute button LED), the hardware monitor might need to know the new device's identifier (some OS APIs allow subscribing to hardware button events). If so, after switching, AudioService would call something like `HardwareAudioMonitor.setDevice(newDeviceId)` to ensure it's monitoring the correct device. This is speculative but worth noting as part of maintaining hardware integration.

External / OS APIs: (Used indirectly through Zoom's code)

- **Windows Core Audio API (WASAPI):** The AudioCaptureService on Windows uses WASAPI endpoints. Switching devices involves: `IMMDeviceEnumerator::GetDevice(deviceId)` to get an `IMMDevice` for the new mic, then creating a new `IAudioClient` and related streams. We won't call this directly; the Zoom audio engine likely wraps it. But the effect is that our `selectMicrophone` ends up making these calls internally. We ensure the device `deviceId` we pass matches the IMMDevice ID string (which Zoom's internal enumeration likely stores).
- **Apple AVFoundation / CoreAudio:** On macOS, switching mic might use `AudioDeviceID` via CoreAudio or AVFoundation's `AVCaptureDevice` for video. The code will call `[AVCaptureDevice deviceWithUniqueID: deviceId]` to get the camera, then reconfigure the capture session. Similarly for audio with `AudioObjectID` selection. Our code ensures to feed the correct unique ID.

- **Linux (ALSA/PulseAudio and V4L2):** For audio, if using ALSA directly, might call `snd_pcm_open` with a device name like "hw:DeviceB". If using PulseAudio, it might use Pulse API to set source. For video, `V4L2` would open `/dev/videoX`. These calls are abstracted but mentionable: we just change the device identifier in whatever open call is made.

- **Permissions APIs:** On first use of a camera or mic, OS might prompt:

- Windows: a user permission for microphone/camera is usually granted at install (there's a system global toggle; Zoom should already have it if the first device worked). No new API call, but if a new device is essentially the same permission category, no new prompt.
- macOS: uses a per-app permission for camera and mic. If Zoom had permission for one camera, it covers all cameras (the permission is category-based, not device-based). So switching cameras typically doesn't prompt again. If it ever did, it's handled by OS; our code just tries to open and OS blocks if not allowed.

- We will utilize the existing permission-check flows – e.g., Zoom might pre-check if `AVCaptureDevice authorizationStatusForMediaType` is authorized. Not specifically part of our new API but relevant to handle failure gracefully if not authorized.

- **Operating System Device Change Notifications:** Zoom likely subscribes to device plug/unplug events (e.g., WM_DEVICECHANGE on Windows, or AVAudioSession route change on iOS, etc.). While not directly our API, our feature could listen to those to automatically update the device lists. For instance, if a user plugs in a new webcam mid-meeting, we'd want it to appear in the menu. AudioService/VideoService handle such events (refresh available devices list and possibly notify UI). We ensure DeviceSelectionMenu can refresh if reopened. This is existing behavior, but we'll verify it works in-call (in some apps, device lists only refresh if you open settings; Zoom often updates the

menu dynamically). We're not writing these OS interfacing code anew – just making sure to call refresh if needed.

Public/External API (Zoom SDK): If Zoom has a developer SDK or external API, we should note if this feature introduces any changes. For example, Zoom might have an API function for SDK users like `ZoomSDKAudioDeviceController::SelectMic(deviceId)`. Our implementation in the client could mirror to that. However, since this spec is about the Zoom client itself, not the SDK, we don't have direct changes to any external API contract except the user-facing interface. So we can state: *No changes to Zoom's external API or meeting protocols are required; this is purely a client-side enhancement.*

In summary, the APIs introduced or utilized align with Zoom's modular structure – new methods in existing classes for selecting devices, plus a new controller interface for the UI to call. These are internal and will be documented for Zoom's development team. Externally, the behavior is surfaced only through the UI, not through any new network or plugin API. The feature mainly extends the **UI-to-service interactions** with a clean interface (DeviceSwitchController) and leverages the **service-to-hardware interfaces** already present. By following this approach, we ensure the feature is implemented with minimal new API surface and maximum reuse of proven calls.

10. Public Interfaces

From the end-user's perspective (and considering UI/UX), the public interface of this feature is the in-meeting device selection controls. Internally, "public interface" can also mean interfaces exposed to other parts of the program or developers. We describe both:

- **In-Meeting UI Elements (User-Facing):** The feature adds *visible controls* in the meeting toolbar:
- The **Microphone button menu** now shows the list of microphones and speakers when clicked [2]. *(If Zoom's UI already had this in some form, we are formalizing it.)* The user can click the **arrow next to the Mute/Unmute button** to reveal:
    - **Select a Microphone** submenu: lists all microphone devices by name. The currently active mic is indicated (e.g., with a checkmark).
    - **Select a Speaker** submenu: (Though our story is about mic/camera, typically Zoom also shows speaker here; we assume speaker switching was already present. We mention it for completeness but our main implementation effort is mic.)
    - Possibly a shortcut to audio settings. This menu interface is intuitive and consistent with Zoom's style. It's essentially the same as the one in Zoom's Settings, but accessible in-call. This **public UI** allows the user to switch without leaving or opening the full settings dialog [6]. No additional steps—just one click to open, one click to select device.
- The **Camera button menu** in the meeting toolbar: Similarly, clicking the arrow next to Start/Stop Video opens a menu:
    - **Select a Camera** list: all available camera devices [4]. Current one is checked. User clicks another to switch.
    - Possibly video settings or virtual background options (existing items). This interface was likely partially present (Zoom had a camera selection in the main UI historically). We ensure it's implemented if it wasn't, or improved if it was disabled.
- **Notification banners or icons:** After switching, if everything is fine, there's usually no persistent public indicator beyond the menu checkmark moving to the new device. If a problem occurs, a **banner message** is a public interface element. For example, if mic fails, a red banner might appear

"Microphone is not working. Zoom is using previous device." or similar. If the user's talking while muted (due to hardware), the banner "You are muted" appears (existing behavior). These notifications are non-modal and appear at top of meeting window for a few seconds – this is how we communicate issues to the user.

- **No new buttons or dialogs** are added, preserving Zoom's clean interface. We integrate into the existing menu system, which is a deliberate choice for consistency.

- **Settings Interface (User-Facing):** In Zoom's Settings (Audio/Video sections, outside of meetings), the user can select default mic and camera. With this feature, any change made in-call is reflected here. For instance, if after the meeting the user opens Settings > Audio, the selected microphone should now be the one they switched to. Conversely, if they had pre-set a certain mic in Settings, that is what shows as default when joining. We ensure these remain in sync. Essentially, the public interface in Settings doesn't change, but it's indirectly updated by this feature's actions. We might add a note in help text like "You can also change devices during a meeting from the menu next to Mute/Video." (Documentation change, not UI element).

- **Programmatic Interface (Internal Modules):** The "public" interfaces that other modules or potential plugins might use:

- While Zoom's client isn't typically open for plugins, if any module needed to trigger a device switch, they would call the **DeviceSwitchController_DSC001** interface (or AudioService/VideoService directly). For instance, an automation script or a test could simulate a device switch by calling those methods. In that sense, we've introduced a clear API internally (`switchDevice`) that is easier than previously (which might have required toggling settings). This is beneficial for maintainability.
- If Zoom had a **command-line or URI interface** (for example, some apps allow `zoommtg://` URIs to do actions), we are not adding anything like that for device switching. It remains manual via UI.
- If Zoom's **SDK** (for developers integrating Zoom in their apps) is considered: the SDK often has an interface for selecting audio devices. We should ensure parity: e.g., `ZoomSDKAudioDeviceController.SetMicDevice(deviceId)` uses the same logic as our feature. The implementation of that SDK call could even route to DeviceSwitchController internally. So from an external developer's standpoint, the *existing SDK API* for switching devices (if it exists) is now more robust (works during calls, not just before call).

- **Network interface:** The feature has no new network calls or server endpoints. It uses the existing meeting stream – we don't inform the server of a "device change" explicitly; the server just continues to receive audio/video. So the meeting protocol (RTP/Zoom's proprietary stream) is unchanged. There is no new public network API message like "ChangeMic" – everything is client-side.

- **Data structures and Config (semi-public):** The configuration file where default devices are stored can be considered a public interface in terms of persistence. We've updated it to record the last used device. This means if an admin or user were to inspect config, they'd see, for example, `audio_input_device: "MicXYZ"`. That's an outcome of this feature. This is not exactly an interface for users, but it's something that persists user's choice.

- **Interaction with other UI flows:** We maintain compatibility. For example, if the user is in a breakout room or if the Zoom UI is minimized, how do they access device switch? Typically the toolbar is

always accessible. No separate interface needed for breakout – it's the same meeting window. If the UI was hidden (minimized), obviously they can't switch until they bring it up. That's fine.

In summary, the **public interface is primarily the enhanced in-call UI controls** (audio/camera selection menus) and the continuation of existing UI notifications for feedback. We aimed to keep the interface intuitive: the user uses familiar controls (the small arrow menus) to do the switching. According to Zoom's design, this is exactly where users would look for such a feature. By not adding new buttons or windows, we ensure a minimal learning curve.

On the internal side, the interface between components (UI to controller to service) is well-defined and follows existing patterns, which acts as the "public interface" for our code within the Zoom app architecture. These have been detailed in the API section and class list, ensuring that other developers working on Zoom can easily find how to invoke or extend this feature without confusion.

11. Data Schemas

This feature does not introduce new complex data models or databases, but it does involve a few data elements related to device configuration and state. We outline changes or uses of data structures in the context of the Zoom client:

- **User Preferences Schema (Configuration Data):** Zoom client maintains a configuration (likely JSON or similar). Relevant fields here include:
- `preferredMicrophone` – stores the device ID (or name) of the user's default microphone. Prior to this feature, users set this via Settings. We confirm that switching the mic in-call will update this field. If previously Zoom only updated it via the settings dialog, we will now also update it when the user switches in meeting. This ensures consistency. The schema might look like:

```
"audio": {
   "mic": { "deviceId": "12345abcd", "deviceName": "Built-in Mic" },
   "speaker": { ... }
}
```

After user switches, `deviceId` and `deviceName` under `"mic"` are updated to the new selection (e.g., "USBMicXYZ"). If the schema did not store deviceName, we might store just ID and look up name in UI each time. Regardless, the ID is the key piece of data.

- `preferredCamera` – similarly, a field for default camera. We introduce or update this. For instance, Zoom might store:

```
"video": {
   "camera": { "deviceId": "camera_dev_2", "deviceName": "HD Webcam" }
}
```

If the user switches camera in-call, we update `"camera.deviceId"` to the new one. This way, next call uses it. If such a field didn't exist, we add it (the settings UI for video selection suggests it likely exists).

- These preferences are saved to disk usually on a change. The UserSettingsManager_SS002 handles writing this out (e.g., calling a SaveConfig).

- **Device Identification Data:** For the switching logic, we rely on identifying devices. Data structures involved:

- **Device List** – AudioService and VideoService maintain a list of available devices. This could be a list of objects with properties:
    - `id` (string or platform-specific identifier)
    - `name` (user-friendly name, e.g., "Microphone (High Definition Audio)")
    - Possibly `index` or `guid` (depending on OS, id might be a GUID on Windows, a UID on Mac, etc.)
    - Possibly `isDefault` flag or type (system default, etc.)
- These lists are likely refreshed at app start and when devices change. The DeviceSelectionMenu uses this list to display entries (mapping each to a menu item).
- The **id** is used as the key for selection. We ensure that the id we store in preferences corresponds to this id field, so that selecting from config or UI is consistent.
- For video devices, we might introduce a similar list if not present. It could be stored in VideoService as `availableCameras: vector<CameraDevice>` where `CameraDevice` has id and name.

- Example (not actual code):

```
struct AudioDeviceInfo {
    std::string id;
    std::string name;
    bool isDefault;
};
std::vector<AudioDeviceInfo> AudioService::micDevices;
std::string AudioService::currentMicId;
```

After switch, `currentMicId` changes and likely corresponds to the stored preference.

- **Runtime State Variables:** Some state is transient but important:

- **Current Active Device IDs** – `AudioService.currentMicDeviceId`, `VideoService.currentCameraDeviceId` as mentioned. These reflect what device is in use *right now* in the meeting. They are updated on switch. They might be initialized from preferences when meeting starts, and updated if user changes via UI. These variables are used by the UI (for instance, to mark the active device in the list, and by the verification modules to know which device to monitor if needed).
- **Mute/Video State Flags** – not new, but reasserting: `AudioService.isMuted` or similar flag remains in place. We do not add new flags for "was muted before switch" because we simply check the current flag. However, the code path might have a local boolean capturing the mute state to decide if it should start the new device right away or not. That's internal logic.
- We ensure the **mute state is not tied to a device** in data – it's a global state. Thus no per-device mute needed (some systems track per-device volumes or mute, but Zoom treats mute at the app

level). The hardware monitor might track per-device hardware mute state if multiple devices have hardware buttons, but typically only the active device's hardware mute matters.

- **Video on/off state** – similarly global (per meeting user), not device-specific. After switching, that flag remains whatever it was.

- **Hardware Device Handles:**

- As mentioned, classes like AudioCaptureService hold a handle or reference to the actual device interface. For example, `AudioCaptureService.currentDeviceHandle` might be an object or pointer to the system's audio stream. Similarly `VideoCaptureService.currentDeviceHandle` to an open camera stream. These are not exposed publicly (hence "internal data"), but from a schema perspective:
    - They might be represented by objects like `AudioDeviceHandle_EXT001` (as seen in class diagram references). This is essentially an abstraction, possibly containing things like a pointer to IMMDevice (Windows) or device index (Linux). When switching, this handle is replaced.
    - The **life-cycle**: old handle closed/disposed, new handle created and stored in the same variable.
- We also note that if hardware monitor AVS003 exists, it may hold a device handle reference to subscribe to events (like "microphone physical button pressed"). That would be updated to the new device's handle too.

- **No new databases or network schema:** This feature is entirely client-side, so no changes to server data schemas. The meeting data (streams) are unchanged format-wise. The client's local data and config as described is the main place for schema adjustments.

- **Logging Schema:** Although not typically mentioned as "data schema," it's worth noting we will log device switch events with relevant info. A log line like:

```
[INFO] 20:35:17 AudioService: Microphone switched from 'Built-in Mic' to
'Yeti USB Mic' (ID:12345abcd)
```

or

```
[ERROR] 20:35:18 VideoService: Failed to switch camera to 'LogitechCam' -
device busy.
```

This is mainly for developers and support, but helps verify behavior. The log format is existing, we just add entries following it.

To summarize, **data schema impacts are minimal and mostly revolve around device identifiers and user preferences**. We extend the Zoom client's config to ensure the chosen devices persist, and we maintain internal variables to track current devices. We reuse the existing data models for devices and

preferences, adding or updating fields where necessary (like adding `preferredCamera` if not already present). This approach avoids complicated migrations or new storage – it's a natural addition to user settings.

The integrity of data is maintained: when a user switches devices, the config and internal state all align to the new device, preventing any mismatch (e.g., we won't have a situation where UI shows one device but audio service uses another, thanks to the single controller update path). The data structures are simple (IDs and names) and reflect exactly what the OS provides, which is reliable. All this ensures that the feature's data aspect is robust and low-risk.

12. Security and Privacy

Changing microphone and camera devices during a call raises some important security and privacy considerations. We need to ensure that the feature does not introduce vulnerabilities and that user privacy (and consent) is respected at all times. Below we discuss these considerations and how our design addresses them:

- **User Consent and OS Permissions:** Accessing a new microphone or camera potentially triggers OS permission checks (especially on first use). It's crucial that the Zoom client only accesses devices the user has permitted.
- *Handling:* Our implementation relies on the user-initiated action (clicking a device from the menu) as implicit consent to use that device. If the OS requires permission (e.g., the user never allowed Zoom to use the webcam before), the OS will prompt the user. We do nothing to bypass or override OS security – the OS dialog must be approved by the user. If denied, we treat it as a switch failure (no access). We maintain that **Zoom's access to mic/cam stays within the bounds granted by the user**. No background or automatic switching to a device the user didn't explicitly choose.
- The device list only shows devices that are present/connected; it doesn't indicate permission status, but if a device is listed, clicking it might prompt a permission if needed (the user will understand this since OS prompt explicitly asks).
- We ensure that after permission is granted, the device starts; if not, we handle as error.
- **No Escalation of Privilege:** The DeviceSwitchController operates within the client's context and doesn't interact with higher-privilege processes. We're not adding any new services or privileged operations. Thus, the risk of this feature being exploited to gain broader system access is minimal – it uses the same calls as normal device usage. We must ensure our code doesn't trust any external input unvalidated:
- The device IDs come either from the OS or config. There's a potential risk if a malicious config file had an invalid device ID (e.g., some injection). However, since device IDs are likely just strings from OS, the worst case an invalid ID will just cause a failure to open (which we handle gracefully). We do not use these IDs in any context like executing a command, so no injection risk.
- We should handle very long or strangely formatted device names/IDs carefully (to avoid buffer overflow). Using std::string in C++ and proper API calls (which expect e.g. a GUID) mitigates that.
- The UI menu is constructed from these strings – we should ensure proper encoding/escaping if any special characters in device names (to avoid UI issues).

- **Maintaining Mute/Camera Off Status (Privacy Guard):** As emphasized earlier, a core privacy guarantee in video calls is that when a user has muted their mic or turned off their camera, the application should **not** transmit audio/video until the user explicitly unmutes or turns video on. Our feature must uphold this:

- When switching devices, we never automatically unmute or start video. The `mute` flag and `videoOn` flag remain unchanged. If muted before, after switching the new mic is **not actively capturing to be sent** (it may be open but audio is discarded or simply not sent because the mute logic in Zoom's pipeline is still active). For camera, if video was off, we might open the camera hardware to be ready (some implementations do this to speed up when user clicks start video) but we **do not send** any video frames until the user hits "Start Video". Many systems also keep the camera LED off if video is off – if we open the device for preview, it might trigger LED on some OSes. That's a consideration:
    - On Windows, opening a camera often turns on the camera light immediately. To avoid surprising the user (camera light coming on while they think video is off), we might decide **not to actually initialize the camera if video is off** (just store selection). This way, the light stays off. Alternatively, if we do open it (perhaps to show a preview in the menu or just to test), we need to communicate that (which could confuse users). It's safer to not engage the hardware until they share video. We will implement accordingly.
- If the user is unmuted, switching mic will result in audio from the new mic being sent, which is intended. If they didn't want to be heard, they should mute first. We can't guess user's intention beyond that. The design at least ensures there's no case where they were muted and end up unmuted inadvertently.

- **Edge privacy scenario:** A user might have relied on a hardware mute on the old device (thinking that's muting them) and stays software-unmuted. When they switch to a new device that has no hardware mute, suddenly audio goes out. This is a user oversight, but our system's response is to quickly alert them via the mute conflict banner. This scenario already exists if a user toggles a hardware mute outside Zoom – the verification feature is the solution. So we continue to rely on that. It's the best we can do since the app can't force carry over a hardware mute state (the new device's hardware state is independent).

- **Camera Privacy:** Zoom should never accidentally show video without user action. By not turning on video when switching cameras (unless it was already on), we maintain that. Also, if a new camera has a higher field of view or shows more of the user's environment, that's something the user implicitly accepts by switching to it. We consider if any **virtual backgrounds or filters** might reset when switching camera (some background processes depend on camera resolution). That's more of a usability thing, but if it did, the user's environment might show briefly. In Zoom, virtual background settings usually persist across cameras if supported. We will test that and ensure the background effect is reapplied after switch (so there isn't a moment where the background is off). This is privacy-related if a user was hiding their room with a background – we don't want the background to drop out during a camera swap. The video pipeline should handle it (likely continuing to apply the selected effect). If not, we must reinvoke the background module after camera init.

- Also, we ensure that if the user had disabled video for privacy and switches camera, we don't do something like accidentally send a one frame or show a preview. The design is to avoid any transmission or UI preview without user clicking "Start Video".

- **Security of Device Access:** We use the OS APIs to open devices. Those calls themselves have security checks (permissions, device locks). We ensure to check return codes and not assume success. There's no memory of previous device's data accessible by new device, etc., so no data leakage. One device's buffers are freed properly before switching to avoid any possibility of mishandling memory.

- We consider if an attacker could exploit the device switching to listen in. If the Zoom client was compromised (malware), it could call our controller to switch to a device like a laptop's built-in mic even if the user had intentionally selected a disconnected mic to avoid sound. But if an attacker can run code in the client, they already can unmute or do worse. So this doesn't add new vector specifically, it's an extension of trusting the client app (which the user does by installing Zoom).

- In terms of safe coding, ensuring that switching logic doesn't overflow, etc., was covered, but also ensure that if multiple devices are switched quickly, resources are not leaked (e.g., not leaving a device handle open each time – that could exhaust OS handles or keep mics on). We explicitly close old devices, so only one mic and one camera are active at a time, which is also a privacy plus (no hidden second mic recording).

- **Network and Data Privacy:** This feature doesn't send any additional data over the network aside from the usual audio/video. We are not transmitting the list of devices or notifying others of the user's device change (other participants have no need to know if you switched mics, they only experience the audio difference). So no privacy concern there. The media stream continues on the same channels, so no renegotiation that might leak info.

- One subtle aspect: If the new camera has a different resolution, Zoom's video engine might renegotiate video quality (send a new SDP or something in its protocol). But that's normal video adaptation. There's no new personal data exposed.

- Device names are not sent out, they stay local (though maybe in logs if user sends log to support, they'll see device names – but that's user-controlled).

- **Logging and User Data:** Device names and IDs might appear in logs (as shown in risk section for debugging). Those logs are only accessible to the user and whoever they send them to (Zoom support). It might reveal the user's device model (e.g., "Logitech C920 webcam"). That's not highly sensitive, but it is a piece of info. This is already something Zoom logs typically, so we follow existing policy (which likely is fine as long as logs aren't shared publicly without user consent). We ensure not to log anything beyond what's necessary (no audio content or such, just device metadata and errors).

- **Testing for Privacy Regressions:** Part of our security plan is to test scenarios specifically related to privacy:

- Ensure that switching while muted never unblocks audio (we simulate switching with mics that have LED indicators to confirm they stay off if should).
- Ensure that if video is off, camera light doesn't come on (we might test with multiple webcams).
- Ensure that the conflict detection triggers properly to warn if needed.

- And check that if a user denies camera permission mid-call (for a new camera), Zoom gracefully handles it without leaking a frame (it should just not start).

- **No Third-Party Data Exposure:** We don't use any third-party services in this feature. All operations are local. So no new privacy policy implications.

In conclusion, the In-Call Device Switching feature is designed with a **privacy-first approach**: it respects the user's mute and camera states, requires explicit user action to switch, and interacts with system resources in a secure, permission-controlled way. By building on Zoom's existing frameworks (which have been vetted for security), we minimize introduction of any new weaknesses. The user remains in control of their audio and video at all times, fulfilling the feature's goal without compromising trust.

---

1 2 4 6 Configuring Your Microphone / Speakers / Camera in Zoom | Canvas @ Yale External Applications | Canvas @ Yale
https://help.canvas.yale.edu/a/1214533-configuring-your-microphone-speakers-camera-in-zoom

3 5 AITool - HW (P1, P2).pdf
file://file-45GS3zCVWKeysR2oFyTsfb