Architectural Blueprint and Implementation Strategy for a Gesture-Based Desktop Control System: A Human-Computer Interaction (HCI) Perspective

1. Introduction: The Paradigm Shift to Natural User Interfaces

The history of computing is fundamentally a history of reducing the abstraction gap between human intent and machine execution. From the punched cards of the mid-20th century to the Command Line Interfaces (CLI) and eventually the Graphical User Interfaces (GUI) that dominate today, each leap has brought the digital world closer to our physical intuition. However, the current dominant paradigm—the WIMP (Windows, Icons, Menus, Pointer) interface—still relies on intermediate mechanical devices like mice and keyboards. These devices, while precise, impose a cognitive load; the user must translate a desire ("turn up the volume") into a specific mechanical sequence (locate specific key, press key).

The project mandate, as defined in the *Gesture-Based Desktop Control System* specification [1], represents a decisive step toward a Post-WIMP era, specifically leveraging **Natural User Interfaces (NUI)**. The objective is to develop a system that allows users to control their desktop environment using customizable hand gestures, captured via a standard webcam. This requires a sophisticated synthesis of computer vision, machine learning, web technologies, and operating system automation.

This report provides an exhaustive architectural analysis and implementation guide for building this system. It addresses the critical requirement of **"UX-Focused Product Design"** [1] by prioritizing latency reduction, visual feedback loops, and error-tolerant interaction models. Furthermore, it details the "Local Bridge" architecture necessary to circumvent browser security sandboxes, enabling a web application to execute privileged desktop actions like Alt+Tab switching and media control.

1.1 The Convergence of Edge AI and Web Technologies

The feasibility of this project rests on recent advancements in Edge AI—specifically, the ability to run high-fidelity computer vision models directly in the web browser using WebAssembly (WASM) and WebGL acceleration. Previously, such systems required heavy, dedicated desktop

binaries (e.g., C++ applications using OpenCV). Today, libraries like **Google MediaPipe** [2] and **TensorFlow.js** [3] enable real-time inference on client devices with zero installation overhead for the vision component.

However, the "zero install" nature of web apps conflicts with the "deep access" required for desktop control. A browser cannot simply "press" the Windows key; if it could, the security implications would be catastrophic. Therefore, this report proposes a hybrid architecture: a "thick" web client responsible for the heavy lifting of vision and logic, communicating via a local loopback network with a "thin" Python agent responsible for execution.

1.2 Scope and Objectives

The core objectives, derived from the project specification, form the pillars of this analysis:

1. **Web-Based Control Panel:** A UI for configuration, feedback, and training.[1]
2. **Customizable Gesture Library:** The ability for users to define new gestures (e.g., "Spock hands") and map them to arbitrary actions.[1]
3. **One-Click Retraining:** A workflow that allows immediate model updates without long training epochs.[1]
4. **Desktop Automation:** Execution of navigation and media commands.[4]
5. **User Experience (UX):** A rigorous focus on learnability, feedback, and error recovery, accounting for 50% of the project's success metrics.[1]

---

2. System Architecture: The Local Bridge Pattern

To satisfy the dual requirements of a "Web User Interface" and "Desktop Control," the system must bridge two isolated environments: the web browser (Sandboxed) and the Operating System (Privileged).

2.1 Architectural Components

The system is composed of three distinct layers, organized in a linear data pipeline that prioritizes low latency.

2.1.1 Layer 1: The Presentation & Inference Layer (Frontend)

This is a Single Page Application (SPA) built with **React.js**. It is responsible for the most computationally intensive tasks.

- **Input:** Raw video stream from the user's webcam.
- **Processing:**
  - **MediaPipe Hands:** Extracts 21 skeletal landmarks in real-time.[5]
  - **Normalization Engine:** Mathematically transforms landmarks to be invariant to scale and position (detailed in Section 3).

    - **TensorFlow.js KNN Classifier:** Performs "One-Click" classification of the normalized landmarks against a user-defined dataset.[3]
- **Output:** High-level intent commands (e.g., GESTURE_DETECTED: "SWIPE_LEFT").

2.1.2 Layer 2: The Transport Layer (WebSocket Bridge)

A persistent, full-duplex communication channel running on ws://127.0.0.1:8000.

- **Rationale:** Standard HTTP requests (REST) introduce significant overhead (TCP handshake, header parsing) for every request. For a gesture system operating at 30 frames per second (FPS), latency must be kept under 100ms to maintain the illusion of instant control.[6] WebSockets allow the server to push status updates back to the UI (e.g., "Command Executed") without polling.

2.1.3 Layer 3: The Execution Layer (Backend)

A lightweight **Python** application using the **FastAPI** framework.

- **Role:** Acts as the "hands" of the system. It listens for intent commands from the WebSocket and translates them into OS-specific input events.
- **Library: PyAutoGUI** is the standard choice here for its cross-platform compatibility (Windows, macOS, Linux) and simple API for keyboard/mouse injection.[7]

2.2 Data Flow Pipeline

The following sequence describes the lifecycle of a single control event, from photon capture to pixel change.

Stage	Component	Action	Latency Budget
1. Acquisition	Browser <video>	Captures webcam frame (640x480).	~16ms (at 60Hz)
2. Inference	MediaPipe (WASM)	Infers 3D landmarks (x, y, z) from the frame.	~20-30ms
3. Logic	TensorFlow.js	Calculates distance to known samples (KNN) and filters noise.	~5-10ms
4. Transport	WebSocket	Transmits JSON	< 5ms (Localhost)

		payload { "action": "NEXT_TAB" }.	
5. Execution	Python/PyAutoGUI	Injects Alt+Tab key codes into the OS event queue.	~10-50ms
Total		**End-to-End Latency**	**~50-110ms**

This budget fits within the 100ms threshold for perceived instantaneous response, a critical heuristic in HCI design.

2.3 Technology Stack Justification

Component	Technology Selected	Alternative Considered	Rationale for Selection
Vision Model	**MediaPipe Hands**	OpenPose / YOLO	MediaPipe is optimized for edge devices (mobile/web) and provides 3D depth, whereas OpenPose is heavier and YOLO requires bounding box conversion.[9]
ML Engine	**TensorFlow.js (KNN)**	Python Scikit-Learn	Running inference in the browser eliminates the need to stream raw video to the Python server, preserving privacy and reducing bandwidth.[2]

Backend	**Python (FastAPI)**	Node.js (Electron)	While Electron is common for desktop apps, Python's ecosystem for automation (PyAutoGUI) and system control is more robust and requires less boilerplate for this specific task.[4]
Protocol	**WebSockets**	HTTP REST / gRPC	WebSockets provide the necessary low-latency, bidirectional capability required for real-time feedback loops.[6]

---

3. Frontend Implementation: Computer Vision and Feature Engineering

The reliability of the system depends entirely on the quality of the data fed into the classifier. Raw data from the camera is noisy and context-dependent. The system must implement a robust feature engineering pipeline to ensure that a gesture performed in the top-left corner of the frame is recognized as identical to the same gesture performed in the bottom-right.

3.1 MediaPipe Hands Configuration and Topology

MediaPipe Hands utilizes a complex pipeline consisting of a **BlazePalm** detector (which finds the hand) and a **Landmark** model (which finds the points inside the hand). It returns 21 landmarks for each detected hand.[5]

The 21 landmarks correspond to specific anatomical points:

- **0:** Wrist
- **1-4:** Thumb (CMC, MCP, IP, Tip)
- **5-8:** Index Finger (MCP, PIP, DIP, Tip)
- **9-12:** Middle Finger
- **13-16:** Ring Finger

- **17-20:** Pinky Finger

Configuration for Desktop Control:

To optimize for the web environment, the GestureRecognizer or HandLandmarker task should be initialized with specific parameters:

- runningMode: "VIDEO": This enables internal temporal filters that smooth landmark movement over time, reducing jitter.[12]
- numHands: 1: Restricting input to a single hand reduces processing load and simplifies the interaction model for desktop control (which is typically a single-pointer paradigm).
- minDetectionConfidence: 0.5: A balanced threshold preventing the hand from vanishing during rapid movement.

3.2 Geometric Invariance: The Mathematics of Customization

The raw output from MediaPipe gives x and y coordinates normalized to the image dimensions (0.0 to 1.0).[5] Using these directly for machine learning is a critical error. If a user trains a gesture while sitting close to the camera, the Euclidean distances between points will be large. If they lean back, the distances shrink, and a distance-based classifier (like KNN) will fail to match the gesture.

To satisfy the "Customizable Gesture Management" requirement [1] robustly, we must normalize the data to be **invariant** to translation, scale, and rotation.

3.2.1 Translation Invariance (Centering)

The absolute position of the hand on the screen is irrelevant for the gesture itself (unless "position" is the trigger, which is rare for static poses). We must shift the coordinate system so the hand's center is always at $(0, 0)$.

We define the **Wrist (Landmark 0)** as the local origin.

$$L_i' = L_i - L_0$$

where L_i is the (x, y) vector of the i-th landmark. This ensures that the wrist is always at $(0, 0)$, regardless of where the hand is on the screen.

3.2.2 Scale Invariance (Size Normalization)

We must account for the distance from the camera. A "large" hand (close) and a "small" hand (far) must mathematically look the same.

We calculate a reference magnitude, typically the size of the palm (distance between Wrist and

Middle Finger MCP).

$$Scale = \max(|L'_i|)$$

$$L''_i = \frac{L'_i}{Scale}$$

This scales the entire hand so that the maximum span falls within a normalized range (e.g., -1.0 to 1.0).[13]

3.2.3 Data Flattening

The final input to the machine learning model is a flattened 1D array. Since we have 21 landmarks, and we typically care about x and y (z is less reliable in monocular webcams), the feature vector size is:

$$21 \text{ landmarks} \times 2 \text{ dimensions} = 42 \text{ floating point numbers}$$

This vector represents the *shape* of the hand, independent of where it is or how big it appears.

3.3 Visualization and Feedback Layer

The "UX-Focused Product Design" [1] requirement necessitates that the user sees what the computer sees. The visualization layer is implemented using an HTML5 Canvas overlaid on the video element.

- **Synchronization:** The canvas draw loop must be synchronized with the requestAnimationFrame of the prediction loop to prevents visual lag (desynchronization between the video hand and the skeleton overlay).
- **Visual Semantics:** The skeleton should change color to communicate state:
  - **White/Blue:** Idle / Tracking.
  - **Green:** Gesture Recognized (Confidence > Threshold).
  - **Red:** Recording / Training Mode.
  - **Yellow:** Low Confidence / Ambiguous state.

---

4. The Machine Learning Engine: One-Click Retraining

The requirement for "One-Click Model Retraining" [1] is the defining technical feature of this application. It demands a model that can learn from a single example (or very few) instantly, without the computational cost of backpropagation or epochs.

4.1 Algorithm Selection: K-Nearest Neighbors (KNN)

While Deep Neural Networks (DNNs) are powerful, they suffer from "catastrophic forgetting"

(learning a new class tends to degrade old ones unless retrained on all data) and require significant computation for training. For this application, **K-Nearest Neighbors (KNN)** is the superior architectural choice.[14]

Why KNN?

1. **Instant Learning:** KNN is a "lazy learner." It does not build a model; it simply stores the training data. "Training" is therefore an $O(1)$ operation (appending to an array).
2. **Few-Shot Capability:** KNN can classify effectively with as few as 10-20 examples per class, whereas a Neural Network might need hundreds.
3. **Transfer Learning:** We are not using KNN on raw pixels. We are using it on the *embeddings* (the 42-float landmark vector) generated by MediaPipe. MediaPipe acts as a powerful feature extractor, and KNN acts as a lightweight classification head.

4.2 Implementing the Training Workflow

The "One-Click" functionality is implemented via the knn-classifier module from TensorFlow.js.[3]

1. **Capture Phase:** When the user clicks "Train" for a gesture (e.g., "Volume Up"), the system initiates a recording burst.
2. **Burst Sampling:** Instead of a single frame, the system captures 30-50 frames over 1-2 seconds. This captures the natural micro-variations of the user's hand holding the pose, creating a dense cluster in the vector space.[17]
3. **Ingestion:**
   ```JavaScript
   // Conceptual Code
   const classifier = knnClassifier.create();
   const activation = tf.tensor(normalizedLandmarks);
   classifier.addExample(activation, "Volume_Up");
   ```
4. **Immediate Inference:** As soon as the examples are added, the classifier can predict against them. No "compile" or "fit" step is needed.

4.3 Persistence and State Management

A critical gap in many web-based ML demos is persistence—reloading the page usually wipes the model. To create a usable desktop control system, the model must be saved.

The knn-classifier does not support standard serialization (saving to file) natively in the same way tf.Model does. We must implement a custom JSON serialization strategy.[18]

Serialization Logic:

1. **Extract Dataset:** Use classifier.getClassifierDataset(). This returns a dictionary of tensors.
2. **Tensor to Array:** Iterate through the tensors and convert them to standard JavaScript

arrays using .dataSync().
3. **Stringify:** Convert the object to a JSON string.
4. **Storage:** Save to localStorage (if small) or IndexedDB (recommended for larger datasets).[18]

Deserialization Logic:

1. Load JSON string.
2. Convert arrays back to Tensors using tf.tensor().
3. Restore classifier state using classifier.setClassifierDataset().

---

5. Backend Engineering: The Python Bridge

The backend serves as the secure gateway to the operating system. It must be robust, secure, and fast.

5.1 Server Architecture: FastAPI

FastAPI is selected for its native asynchronous capabilities.[11] Traditional synchronous frameworks (like standard Flask) would block the thread while waiting for a PyAutoGUI command to finish (which often involves sleep timers). FastAPI allows handling the WebSocket connection on one thread while offloading execution tasks, ensuring the "ping" from the frontend is never delayed.

WebSocket Lifecycle:

1. **Handshake:** Frontend connects to ws://localhost:8000/ws.
2. **Loop:** The server enters an await loop, listening for messages.
3. **Parsing:** Messages are expected in JSON format: {"command": "volume_up", "confidence": 0.98}.
4. **Execution:** The server invokes the automation logic.

5.2 Desktop Automation: PyAutoGUI

PyAutoGUI is the standard library for programmatic GUI control.[7] However, it has specific behaviors across different Operating Systems that must be handled.[8]

5.2.1 Operating System Differences

- **Windows:** Standard keys like win, alt, ctrl work as expected.
- **macOS:** The "Command" key is ['command'] or ['cmd']. The alt key is often mapped to option. Tab switching is command + tab, whereas Windows is alt + tab.
- **Linux (X11/Wayland):** PyAutoGUI uses Xlib. Wayland support is limited and may require specific environment configurations or fallback to uinput.

Implementation Strategy:

The Python backend should detect the OS at runtime (platform.system()) and load the appropriate key mapping dictionary.

Python

```
# Conceptual OS Mapping
import platform
SYSTEM_OS = platform.system()

KEY_MAP = {
    "switch_tab": ["alt", "tab"] if SYSTEM_OS!= "Darwin" else ["command", "tab"],
    "volume_up": "volumeup",  # PyAutoGUI handles media keys mostly universally
    "copy": ["ctrl", "c"] if SYSTEM_OS!= "Darwin" else ["command", "c"]
}
```

5.2.2 Fail-Safes and Latency

PyAutoGUI includes a fail-safe: dragging the mouse to a corner of the screen throws an exception to kill the script. This should be kept active during development but managed in production to avoid crashing the server.

- **Latency Optimization:** By default, PyAutoGUI adds a 0.1s pause after every command. For smooth media control (e.g., volume adjustments), this delay creates a sluggish feel. We must set pyautogui.PAUSE = 0.01 or 0 for specific repetitive actions.[20]

5.3 Security Considerations

Running a WebSocket server that accepts input commands creates a potential attack vector.

- **Localhost Binding:** The server must explicitly bind to 127.0.0.1, *not* 0.0.0.0. This ensures it is not accessible from other devices on the network.[21]
- **Command Whitelisting:** The server should *never* execute arbitrary code sent from the frontend (e.g., exec()). It should only accept specific pre-defined tokens (IDs) that map to hardcoded functions on the server side.
- **Token Authentication:** Ideally, the connection should require a handshake token to prevent malicious websites from scanning localhost ports and injecting commands (Cross-Site WebSocket Hijacking).[22]

---

6. UX/UI Design Strategy: Patterns for Machine Learning

Designing for AI requires different patterns than traditional software. The user is not just operating the system; they are *teaching* it. This creates a "Gulf of Evaluation" where the user wonders, "Did the system understand me?"

6.1 The "Wizard" Pattern for Gesture Recording

To satisfy the "Guided Process" requirement [1], the gesture creation workflow should follow a strict, linear **Wizard Pattern**.[23]

1. **State 1: Instruction.** "Position your hand for the 'Volume Up' gesture."
2. **State 2: Countdown.** "Recording in 3... 2... 1..." (Allows the user to stabilize their hand).
3. **State 3: Data Ingestion.** A progress ring fills up. The UI shows the skeleton turning **Red** to indicate recording. The system captures ~50 frames.
4. **State 4: Verification (The Feedback Loop).** The system enters a "Test Mode." The user is asked to perform the gesture again. A **Confidence Meter** (Gauge or Bar) reacts live. This is crucial: it proves to the user that the model works *before* they bind it to an action.
5. **State 5: Binding.** A dropdown menu allows selecting the action (e.g., "Volume Up").

6.2 The "Midas Touch" Problem and Background Class

A major trend in gesture recognition UX is the "Midas Touch" problem: the system interpreting every movement (scratching a nose, drinking coffee) as a command.

- **Solution: The "Null" Class.** The system must explicitly train a "Nothing" or "Background" class. The user should be guided to record their hand in a relaxed state or empty background.
- **Solution: Activation Thresholds.** Actions should only trigger if the confidence score > 90% for a sustained duration (e.g., 5 frames). This "debouncing" prevents jittery firing of commands.[24]

6.3 Visual Feedback Hierarchy

The UI must communicate system status at a glance:

- **Webcam Feed:** The "Mirror" is central. Users must see themselves to align their hand.
- **Skeleton Overlay:**
  - **White:** Tracking active, no gesture detected.
  - **Green:** Gesture detected (Action Triggered).
  - **Red Pulse:** Recording in progress.
- **Toast Notifications:** When an action triggers (e.g., "Mute Toggled"), a non-intrusive toast should appear. This bridges the gap between the web app and the desktop effect.

7. Performance and Latency Optimization

For the system to feel responsive, the total latency loop must be minimized.

7.1 WebGL Acceleration

TensorFlow.js and MediaPipe must be configured to use the webgl backend. This utilizes the GPU for matrix operations. The CPU backend is significantly slower and will result in FPS drops, making the gesture tracking feel "laggy."

7.2 Garbage Collection

High-frequency loops (30 FPS) in JavaScript can create massive amounts of garbage (temporary tensors).

- **Optimization:** Use tf.tidy() to wrap inference blocks. This automatically cleans up intermediate tensors. Failing to do so will cause a memory leak that crashes the browser tab within minutes.[25]

7.3 Concurrency Models

- **JavaScript:** The inference loop acts on the main thread (mostly). Using requestAnimationFrame ensures it doesn't block UI rendering.
- **Python:** Using async/await in FastAPI is critical. Standard synchronous code would pause the WebSocket heartbeat while PyAutoGUI executes a key press, potentially causing the connection to time out.[26]

8. Implementation Roadmap and Testing Strategy

Phase 1: The Vision Core (Frontend)

- **Goal:** Render webcam feed and draw the skeleton with 30 FPS.
- **Key Tech:** React, MediaPipe Hands.
- **Test:** Verify landmarks are accurate and do not flicker.

Phase 2: The Logic Core (ML)

- **Goal:** Implement KNN, normalization, and the "Add Example" logic.
- **Key Tech:** TensorFlow.js knn-classifier.
- **Test:** Train "Open Hand" vs "Fist" and verify classification stability. Ensure JSON save/load works.

Phase 3: The Bridge (Backend)

- **Goal:** Establish WebSocket comms and basic OS control.
- **Key Tech:** FastAPI, PyAutoGUI.
- **Test:** Send {action: "volume_up"} via Postman/wscat and verify system volume changes.

Phase 4: Integration and UX

- **Goal:** Connect Frontend to Backend. Build the Wizard UI.
- **Key Tech:** React State Management, WebSocket hooks.
- **Test:** Full end-to-end latency test. User acceptance testing for the "One-Click" workflow.

---

9. Conclusion

The **Gesture-Based Desktop Control System** is a sophisticated interplay of modern web capabilities and desktop automation. By strictly adhering to the architectural separation of concerns—using the browser for what it does best (Vision/ML) and the desktop agent for what it does best (Execution)—we can achieve a system that is both powerful and secure.

The integration of **MediaPipe's** efficient landmarks, **TensorFlow.js's** transfer learning capabilities, and **Python's** automation ecosystem creates a "Local Bridge" that satisfies all functional requirements. However, the ultimate success of the project relies on the **UX implementation**: specifically, the normalization mathematics that make gestures robust, the "Wizard" workflows that make training intuitive, and the visual feedback loops that reassure the user. This report provides the blueprint to build not just a prototype, but a polished, responsive, and truly customizable product.

10. Appendix: Data Tables and Reference Material

Table 1: Comparative Analysis of ML Approaches for Gesture Recognition

Feature	**K-Nearest Neighbors (KNN)**	**Multi-Layer Perceptron (MLP)**	**Long Short-Term Memory (LSTM)**
Training Speed	**Instant (Zero-Shot)**	Slow (Epoch-based)	Very Slow
Data Required	**Low (10-50 samples)**	Medium (100s)	High (1000s)
Complexity	Low	Medium	High
Use Case	**Custom/User-Defi**	Pre-trained Static	Dynamic/Time-Seri

	ned Gestures	Gestures	es Gestures
Browser Load	Light (Memory Intensive)	Medium (CPU Intensive)	Heavy

Table 2: PyAutoGUI Key Mapping Strategy

[8]

Action	**Windows Key Code**	**macOS Key Code**	**Linux Key Code**
Switch Tab	['alt', 'tab']	['command', 'tab']	['alt', 'tab']
Volume Up	'volumeup'	'volumeup'	'volumeup'
Play/Pause	'playpause'	'playpause'	'playpause'
Copy	['ctrl', 'c']	['command', 'c']	['ctrl', 'c']
Close Window	['alt', 'f4']	['command', 'w']	['alt', 'f4']

Table 3: MediaPipe Hand Landmark Indices (Reference)

[5]

Index	**Anatomical Point**	**Function in Normalization**
0	**Wrist**	**Origin Point (0,0)** for Translation Invariance.
9	**Middle Finger MCP**	**Reference Point** for Scale/Rotation Calculation.
4	Thumb Tip	Used for Pinch detection.
8	Index Tip	Used for Pointing/Clicking.
12	Middle Tip	Used for Swipe tracking.

Table 4: Latency Optimization Checklist

Layer	Optimization Strategy	Expected Gain
Vision	Use modelComplexity: 1 (Lite)	+10-15 FPS
Vision	Use minDetectionConfidence: 0.5	Reduced jitter
ML	Use tf.tidy() for tensor cleanup	Prevents memory leaks/stalls
Network	Bind WebSocket to 127.0.0.1 (Localhost)	< 1ms latency
Backend	Set pyautogui.PAUSE = 0.01	+90ms responsiveness per action
Backend	Use async WebSocket handlers	Prevents blocking on I/O

Works cited

1. Gesture-Based Desktop Control System.pdf
2. Gesture recognition guide for Web | Google AI Edge, accessed February 19, 2026, https://ai.google.dev/edge/mediapipe/solutions/vision/gesture_recognizer/web_js
3. Gesture Recognition Using Tensorflow.js - GetStream.io, accessed February 19, 2026, https://getstream.io/blog/tensorflow-gesture-recognition/
4. Control a desktop app through web with WebSockets | by François Voron - Medium, accessed February 19, 2026, https://medium.com/@fvoron/control-a-desktop-app-through-web-with-websockets-41626d949e3b
5. layout: forward target: https://developers.google.com/mediapipe/solutions/vision/hand_landmarker title: Hands parent: MediaPipe Legacy Solutions nav_order: 4 — MediaPipe v0.7.5 documentation - Read the Docs, accessed February 19, 2026, https://mediapipe.readthedocs.io/en/latest/solutions/hands.html
6. FastAPI + WebSockets + React: Real-Time Features for Your Modern Apps -

Medium, accessed February 18, 2026, https://medium.com/@suganthi2496/fastapi-websockets-react-real-time-features-for-your-modern-apps-b8042a10fd90
7. How To Automate Your Desktop With PyAutoGui - YouTube, accessed February 18, 2026, https://www.youtube.com/watch?v=RMdN3Vq7sTE
8. Keyboard Control Functions - PyAutoGUI documentation - Read the Docs, accessed February 19, 2026, https://pyautogui.readthedocs.io/en/latest/keyboard.html
9. Dynamic Hand Gesture Recognition Using MediaPipe and Transformer - MDPI, accessed February 18, 2026, https://www.mdpi.com/2673-4591/108/1/22
10. How to Train Custom Hand Gestures Using Mediapipe - Instructables, accessed February 19, 2026, https://www.instructables.com/How-to-Train-Custom-Hand-Gestures-Using-Mediapipe/
11. WebSockets - FastAPI, accessed February 19, 2026, https://fastapi.tiangolo.com/advanced/websockets/
12. Gesture recognition task guide | Google AI Edge, accessed February 19, 2026, https://ai.google.dev/edge/mediapipe/solutions/vision/gesture_recognizer
13. How to Normalize Hand Landmark Positions in Video Frames Using MediaPipe?, accessed February 19, 2026, https://stackoverflow.com/questions/78329439/how-to-normalize-hand-landmark-positions-in-video-frames-using-mediapipe
14. A Comparative Analysis of CNN and SVM for Static Sign Language Recognition Using MediaPipe Landmarks Journal of Intelligent Syst, accessed February 18, 2026, https://journal.unesa.ac.id/index.php/jistel/article/download/40750/13813/141318
15. Recommended ML model for static and dynamic hand gesture recognition? - Reddit, accessed February 18, 2026, https://www.reddit.com/r/MLQuestions/comments/1p2zrw3/recommended_ml_model_for_static_and_dynamic_hand/
16. Hand Pose Training with TensorFlow.js and ml5.js - GitHub, accessed February 18, 2026, https://github.com/GarikHarutyunyan/js-hand-pose-training
17. Transfer Learning with KNN - Processing with AI, accessed February 19, 2026, https://processing-with-ai.gitlab.io/part2/transfer-learning/
18. Add save() and load() methods to knn-classifier in tfjs-models · Issue #633 - GitHub, accessed February 19, 2026, https://github.com/tensorflow/tfjs/issues/633
19. LocalStorage vs IndexedDB: Choosing the Right Solution for Your Web Application, accessed February 19, 2026, https://shiftasia.com/community/localstorage-vs-indexeddb-choosing-the-right-solution-for-your-web-application/
20. Cheat Sheet - PyAutoGUI documentation - Read the Docs, accessed February 19, 2026, https://pyautogui.readthedocs.io/en/latest/quickstart.html
21. Full Duplex Communication Between Web App and Desktop Application Using WebSocket, accessed February 18, 2026, https://dev.to/mark0960/full-duplex-communication-between-web-app-and-des

ktop-application-using-websocket-51gd
22. How to Implement WebSockets in FastAPI - OneUptime, accessed February 19, 2026, https://oneuptime.com/blog/post/2026-02-02-fastapi-websockets/view
23. Guided Tour design pattern - UI-Patterns.com, accessed February 19, 2026, https://ui-patterns.com/patterns/Guided-tour
24. Tensorflow.js Series. KNN Classification. | by Practicing DatScy | CodeX - Medium, accessed February 19, 2026, https://medium.com/codex/tensorflow-js-knn-usage-19585f32eecb
25. Make a Webcam Controlled Game with TensorFlow.js - Codédex, accessed February 19, 2026, https://www.codedex.io/projects/make-a-webcam-controlled-game-with-tensorflowjs
26. Send / receive in parallel using websockets in Python FastAPI - Stack Overflow, accessed February 19, 2026, https://stackoverflow.com/questions/67947099/send-receive-in-parallel-using-websockets-in-python-fastapi