ChatGPT

Executive Summary

- **ShuttleSense Plan:** Use a fast, lightweight pose model (e.g. MoveNet or MediaPipe BlazePose) on a single fixed camera to capture 2D keypoints of the athlete in real time [1]. Smooth the pose data with filters (OneEuro/Kalman) to reduce jitter [2] while preserving responsiveness.
- **Rule-Based Event Detection:** Implement a finite-state machine (FSM) that uses joint thresholds and hysteresis to detect footwork events (split steps, lunges, recoveries) reliably before introducing ML classifiers. This ensures low-latency, accurate feedback for quick movements and minimizes false positives under dynamic conditions [3].
- **One Correction at a Time:** Design the feedback system to surface one coaching cue at once, using cooldown periods to avoid overload. This mirrors best coaching practices (e.g. "restrict yourself to one cue at a time for maximum effect" [4]) and improves user focus and retention.
- **Privacy-First Data:** By default, only store processed pose keypoints and labeled events/timestamps – not raw video. Representing sessions as abstract pose sequences preserves user privacy [5] while still capturing essential movement metrics. Raw video can be optional (for user's playback) and can be discarded after processing.
- **Evidence-Based Coaching:** Ground all feedback in measured data. The system logs quantitative metrics (e.g. stance width, time to recover) with timestamps, and the AI coach chat uses retrieval to pull these facts. The chat will **never** invent advice beyond the session JSON, instead citing specific moments ("e.g. at 02:10 you paused 1.5s before returning to base [5]"). This prevents generic or hallucinated tips.
- **Existing Tools Leverage:** Use open-source libraries to accelerate development. For pose detection, established solutions like MediaPipe or MoveNet offer real-time performance on phones [1]. Smoothing filters (OneEuro [6]) and pose-analysis toolkits (e.g. OpenMMLab MMPose [7]) are available. Repos for fitness coach apps (e.g. AI exercise form checkers) provide reference implementations for rep counting, feedback UIs, and even voice cues (e.g. RepDetect uses MediaPipe on Android with voice feedback).
- **Architecture Overview:** For V1, a simple web app where users upload a video for cloud processing is recommended. A Cloud Run service can run the pose model on each frame (~18k frames for 10min at 30fps) using a GPU for speed. The pipeline: video → pose frames → smoothing → event detection → JSON report → visualization + chat. Expect ~3–5 minutes processing per 10-min video on a modest GPU (cost just a few cents per session). All heavy compute stays server-side, keeping the client lightweight.
- **Future Roadmap:** Gradually improve accuracy and depth of analysis. Next steps include camera calibration (using court lines or a calibration pose) to better estimate real-world distances, personalizing thresholds to the user's baseline, and perhaps multi-angle capture (two phones) for true 3D analysis. However, complex features like shuttle tracking, stroke recognition, or multi-camera fusion should be **deferred** until the core footwork coach is robust – these are tempting but error-prone additions that require extensive data and add latency.
- **Immediate Next Step:** Start a pilot implementation focusing on pose extraction and one key event (e.g. lunge detection) to validate the end-to-end flow with real user footage.

A. Best Pose Stack for Single-Angle Video Analysis

A1. Pose Estimation Options (Speed vs Accuracy)

For badminton's quick footwork and lunges, we need a pose model that balances **speed, accuracy, and foot/keypoint reliability**. Modern light-weight models make real-time pose estimation feasible even on mobile hardware [1]:

- **MediaPipe BlazePose (Full-body, 33 keypoints):** BlazePose (available via Google's MediaPipe Solutions and ML Kit) detects full-body landmarks including feet (toes, heels) in real time. It's optimized for mobile/edge with a lightweight convolutional architecture and even supports multi-person. MediaPipe's pose model is known for stability due to "virtual keypoints" and built-in temporal filtering; it can achieve ~30fps on modern phones (even higher on laptops) [1] [8]. *Readiness:* Production-proven in many fitness apps. *Why useful:* Foot keypoints allow detecting foot placement and airborne vs grounded feet (useful for split-step hops). *Limitations:* May struggle with very fast motions or occlusion – e.g., legs blurred during a quick lunge might cause momentary keypoint swaps. Also, it requires integration via MediaPipe's API (or ML Kit on Android), which can add some complexity but provides cross-platform support.

- **MoveNet (Google, 17 keypoints COCO format):** A very fast pose model offered in two versions – Lightning (smaller, ~5ms/frame) and Thunder (larger, more accurate). MoveNet Lightning has been demonstrated to run **well above 30fps on phones** [1], making it ideal for live feedback. *Readiness:* Production-ready (used in Google's example apps for yoga, etc.). *Why useful:* Ultra low latency; straightforward TensorFlow Lite or TF.js deployment. *Limitations:* Only 17 keypoints (no explicit foot tip markers – it has ankles but not toes). That means subtle foot placements (heel vs toe lift) are not distinguishable, though for footwork drills, ankle positions + velocity might suffice to infer jumps or stance width. MoveNet can also lose track during occlusions (e.g., if legs cross).

- **YOLOv8 Pose (Ultralytics):** A newer one-stage model that extends YOLO object detection to output pose keypoints. The smallest variant (YOLOv8n-Pose) is compact (~4 million params) and **achieves ~66 FPS on GPU** [9], implying it could run real-time on moderate hardware. It's been applied in sports contexts – a 2025 study customized YOLOv8-Pose with local attention and **built a badminton pose dataset**, reporting superior accuracy on badminton-specific movements [10] [11]. *Readiness:* Cutting-edge, with an easy Python API via Ultralytics. *Why useful:* Unified detection+pose (robust if multiple people or if needing to track the player among others), and proven on badminton data with fast inference [9]. *Limitations:* The Ultralytics code is GPL-3 licensed (restrictive for closed-source apps). To avoid license conflict, one can use the exported model (ONNX/CoreML) with a custom inference loop (not using Ultralytics code). Also, YOLO's pose uses COCO 17 keypoints (no toes), similar limitation as MoveNet.

- **OpenPose and Derivatives (AlphaPose, etc.):** OpenPose (CMU) was the pioneer of full-body pose (including hands and feet keypoints) and is very accurate. AlphaPose and others build on similar architectures (often using heavier backbones like HRNet). *Readiness:* Academic/experimental – OpenPose can run in real time on a high-end GPU, but on mobile or web it's far too slow (and its license disallows commercial use without a hefty fee [12]). *Why useful:* Provides foot keypoints and even finger details; good for offline analysis if accuracy trumps speed. *Limitations:* **Not suitable for low-latency** – e.g., HRNet-based models can hit ~15 FPS even on strong GPUs [13]. Also, licenses (OpenPose and AlphaPose are non-commercial by default [14] [12]) are a show-stopper for a product. We will avoid these for ShuttleSense V1.

Recommended Pose Backbone for V1: MoveNet Lightning is a top choice for real-time drill feedback given its speed and proven deployment on phones [1]. It reliably captures the major joints needed for footwork metrics (hips, knees, ankles). For the video analysis (cloud) scenario, we can afford a slightly heavier model for a bit more accuracy – e.g. MediaPipe's "Full" BlazePose model or MoveNet Thunder – since a few seconds of processing delay is acceptable for post-session analysis. Both BlazePose and MoveNet have high accuracy on typical poses; BlazePose might handle extreme limb positions slightly better (due to its 33 keypoints and training on diverse fitness/yoga data) [8]. On balance, **we will use MediaPipe Pose (Full) on the cloud for robust analysis, and MoveNet Lightning on-device for any live mode.** This combination covers both needs and shares similar coordinate systems (we can standardize on COCO-format keypoints subset for simplicity).

A2. Pose Smoothing & Tracking Stability

Raw pose keypoint data tends to be noisy frame-to-frame – causing jittery skeleton limbs that could trigger false event detections. **Smoothing filters are essential** for stable metrics [2]. We prioritize techniques that introduce minimal lag:

- **OneEuro Filter:** A lightweight adaptive low-pass filter ideal for real-time interaction. It dynamically adjusts cutoff frequency based on signal velocity – smoothing noise when movements are slow, but allowing quick changes to pass through with minimal delay [6]. This filter has been used in VR/AR and pose tracking contexts to combat jitter while retaining reactivity. *Implementation:* We can apply a 1€ filter independently to each keypoint (x, y coordinates), or to derived signals (e.g., joint angles or distances) to smooth higher-level metrics. Tuning is required (parameters: min_cutoff and beta [15]) – e.g., for badminton, we'd set a higher min_cutoff for feet since they move rapidly (to not lag behind quick steps), but a lower cutoff for torso angles to reduce camera jitter effect.

- **Exponential Moving Average (EMA):** Simpler smoothing with a fixed smoothing factor. Easy to implement (just a weighted sum of previous value and new value). We can use a short window EMA for joints like ankles to smooth minor oscillations. However, a fixed EMA can introduce noticeable lag if too smooth, or leave jitter if too responsive – not as flexible as OneEuro.

- **Kalman Filter:** If we model the motion of keypoints (position and velocity), a Kalman filter can predict and correct keypoint positions each frame, using the model's confidence as measurement noise. This can stabilize occlusions (e.g., if a foot is briefly occluded by the body, the filter can "coast" the last known position instead of jittering). Some open-source solutions like FilterPy (Python) or OpenCV's Kalman classes can be leveraged. *However*, Kalman tuning (noise covariances) can be tricky without a physical motion model. It's likely overkill for an initial solution, but worth exploring for smoothing 1D signals like stride timing.

- **Built-in Smoothing from Pose Trackers:** MediaPipe's Pose Landmarker API internally has smoothing if enabled (it uses an IIR filter over landmarks). Google ML Kit Pose on Android also offers a stabilized "accurate" mode that trades speed for stability [16] [17]. We could simply use those out-of-box smoothing options initially, then layer our own filter if needed for specific signals.

Best Practice: We will start with the **OneEuro filter** on key joint coordinates because of its proven balance between jitter reduction and latency (designed for "high precision & responsiveness" [6]). For example, apply OneEuro to the ankle positions to smooth the rapid foot taps, and to knee angles to stabilize knee measurements. We'll augment this with simple logic: if a keypoint's confidence/visibility from the model is low or drops (MediaPipe provides a visibility score per landmark [18] [19]), we can

momentarily decrease the filter cutoff (essentially freezing it a bit) to avoid wild jumps from spurious data [20]. The output of the smoothing stage will be a sequence of stable poses, which feed into the event detection.

A3. Joint Angle & Metric Reliability

To give meaningful feedback, we need to compute domain-specific metrics from 2D joint data, while accounting for the single-camera perspective:

- **Stance Width:** Measured as distance between the feet. In 2D, we can use the distance between left and right ankle keypoints (or a combination of ankle and toe if available). For a camera roughly behind or in front of the player, the horizontal component of this distance corresponds to actual lateral stance. We will normalize stance width relative to the player's **shoulder width** (distance between left and right shoulders) to personalize it. This ratio helps flag too-narrow or too-wide stances independent of player size. *Reliability:* If the camera is at an angle (e.g. 45°), the apparent foot distance can shrink when one foot is forward – causing underestimation. We assume the user's camera angle is roughly behind or slightly to the side. In later iterations, we can correct via calibration (e.g., using known court line width as reference to derive scaling or homography [21]). For V1, a heuristic approach is to detect when one foot is much closer to camera than the other (perhaps by relative ankle **y** positions or foot visibility) and in those cases, treat stance width measurements with caution or exclude those moments.

- **Knee Collapse (Valgus) Indicator:** A key injury-related form issue is the inward collapse of the knee during lunges or landings. From a frontal or rear view, this would show up as the knee moving towards the midline relative to the foot. We can approximate this by the 2D angle at the knee (hip–knee–ankle angle in the frontal plane projection). If the knee angle (as seen by camera) drops below, say, 160° (indicating a not straight alignment) at a moment of landing, and if the knee joint's x-coordinate is significantly inside of the ankle's x-coordinate (for instance, if left knee is to the right of left ankle in the image for a left leg lunge), that could signal valgus collapse. *Caveat:* With a single angle, depth ambiguity can mislead this measurement – a mild valgus might look severe if the camera is not perfectly aligned. We will set conservative thresholds and mark such detections as low confidence warnings, potentially asking the user to provide a more frontal video if this is a focus area.

- **Torso Lean & Balance:** We can derive torso angle from the line between hips and shoulders. Excessive forward lean during a lunge might indicate reaching too far or off-balance posture. If the torso is past, say, 30° off vertical at lunge impact, we flag it as a balance issue. Similarly, we monitor how much the torso or head "wobbles" after landing – e.g., variance in shoulder position in the second after landing – as a proxy for stability. These signals will be qualitative at first.

- **Split Step Timing:** In a real match, "split step timing" is measured as the interval between opponent's hit and the player's landing from the split step. In our single-player scenario, we simulate this either with an audio cue or within self-directed drills (e.g., user does split-step then immediately moves). For now, our metric can be the consistency of split step timing relative to the start of each footwork sequence. For instance, if the user always does a small hop and lands just as they push off to a corner, that's ideal. If we detect the push-off (first large movement) often precedes the split hop landing, they might be "late on the split." We'll measure: time difference between the **split step landing** and the subsequent **movement initiation**. If negative (already moving before finishing split) or very large, it's suboptimal. Without an external reference, this is a rough proxy but still actionable in drills.

- **Recovery Speed:** After a lunge to a corner, how quickly does the player return to the "base" position (usually center of court or start point)? We can measure the time from the moment of hitting maximum lunge (deepest point or racket swing, if detectable) to the moment both feet come back within a small radius of the starting stance. The baseline "good" might be, say, <1 second. This requires identifying base position – we'll approximate base as the median hip position at the start of the drill (assuming they begin at center). Then track when the hips (or feet) come back near that area.

Handling Unreliable Signals: Fast badminton motions and single-camera 2D data mean some metrics will be noisy. Occlusion is a major issue – e.g., from a behind angle, a deep lunge might hide one leg entirely. Our approach: use the model's confidence scores (visibility) to decide when to trust a metric. If an event happens during an occlusion (like a split jump where knees occlude ankles briefly), we rely on the other foot or skip that instance. We'll note such uncertainties in the report, rather than give a potentially wrong correction. In sum, we prioritize **precision over completeness** – better to occasionally miss detecting a split step than to wrongly accuse the user of a technique flaw that didn't happen.

B. Footwork Event Detection from Pose (Rules-First Approach)

We will detect a set of **key footwork events** in the pose timeline using deterministic rules. Each event corresponds to a phase or action in footwork: e.g. *Split Step*, *Push-off*, *Lunge (Reach)*, *Shot Execution*, *Recovery Step*. By breaking the movement into these events, we can attach specific feedback to each (timing, form, etc.). The detection logic will use thresholds on joint positions/velocities, with hysteresis and state memory to ensure stability.

B1. Events of Interest and Detection Rules

- **Split Step (Hop) Detection:** The split step is typically a small vertical hop just before moving. In pose terms, both feet leave the ground briefly and land almost together. To detect: monitor the ankles' vertical velocity or distance to ground. With a single camera, we can't directly measure actual height off ground, but we can infer it if the camera is at a fixed position: a rapid drop in both ankle y-coordinates followed by a rise indicates an upward motion (or simply detect the *peak* of hip height – the hop causes a momentary rise in the center of mass). Concretely, if both ankles *simultaneously* go **up** by more than a small epsilon (relative to their recent baseline) and then down within ~0.2s, that pattern indicates a hop. Another cue: knees will quickly bend then extend. So, a sequence of knee angle decreasing then rapidly increasing can corroborate. **Rule:** When in "ready" state, if knee angle drops > X (e.g. >15° bend) and then rebounds and ankles show a small upward motion (dy < -threshold) within a short window, mark a Split Step event. Use a short cooldown (maybe 0.5s) to avoid double-trigger. *Hysteresis:* We require both feet to have landed (i.e., reached their lowest y after the hop) before allowing another split detection, ensuring one hop is counted once.

- **Push-Off / Direction Change:** Right after the split, the player pushes in some direction (forward/back/side). We detect the **first foot movement direction** post-split. E.g., if the next significant motion is to the right, the right foot might push outward. We can detect which foot moves first: measure sudden increase in distance between feet or a foot's displacement. But practically, this is rolled into recognizing the lunge. We might not label "push-off" separately in the report, but we'll use it internally: once split step is done, note which direction the center of mass (hips) starts moving – that indicates the direction of the footwork sequence.

- **Lunge (or step to corner) Execution:** A lunge is characterized by a large stride and knee bend on the leading leg. Detection rule: when distance between feet exceeds a threshold (e.g. >150% of shoulder width) and front knee angle goes below a threshold (e.g. < 120°), that indicates a lunge position. We also look for one foot ahead of the body: e.g., the front ankle significantly ahead (in camera view) of the back ankle. In a rear camera view, a lunge to forward-right corner might show the right ankle far ahead (downwards in image) of the left. We will have heuristics per direction if needed (though for now, any big stride qualifies). **Rule:** If currently not in a lunge and foot distance suddenly increases beyond LungeThreshold and one knee is deeply bent, trigger a Lunge event at that moment. We can refine to classify *which corner/direction* based on whether it was left or right foot and forward/back – but V1 can simply record "Lunge" with perhaps a note "(to backhand side)" etc. *Hysteresis:* Once in "lunge" state, we don't trigger another until the player recovers (feet come closer).

- **Shot or Peak Moment (optional):** In a full analysis, we might want the moment of racket swing or impact. However, with just pose, recognizing the shot contact (especially without a racket visible clearly) is unreliable. We will skip explicit "shot" detection in V1. If needed, we can approximate the apex of the lunge as the shot moment.

- **Recovery to Base:** Detection rule: after a lunge, once the player starts moving back and their feet return within a certain distance (e.g. back to ~shoulder width or the original stance location), mark a Recovery event. Essentially, when a "lunge" state ends – i.e., foot distance drops back below a threshold and both feet are on or around the starting position (if we know it). This could be a continuous process, but we'll define a single event at the moment they are *back and ready*. Recovery time is measured from lunge event to this point.

- **Split Step (Subsequent) or Next Rally Event:** After recovery, the cycle might repeat for another shot or move. We set the state machine to "ready" again and look for the next split or lunge. In drills, the user might do a sequence of lunges to different corners; in that case, each change of direction might involve a mini-split or bounce. We handle them similarly.

B2. Finite-State Machine Design

We implement the above with a simple **state machine** that transitions through phases and thereby imposes temporal logic (to avoid e.g. detecting a lunge before a split in the timeline):

States might include: **Ready -> SplitHop -> Moving -> Lunging -> Recovering -> Ready (again)**.
- **Ready:** expecting a split or direct movement.
- **SplitHop:** detected a split hop, waiting for landing completion and direction push.
- **Moving:** after split, moving toward a shot – might be running or sidestepping, culminating in a lunge.
- **Lunging:** in an extended stance – waiting for recovery.
- **Recovering:** moving back to base. Once complete, go to Ready if drill continues.

The FSM will use the above rules to trigger transitions. For example: In Ready, if split-hop condition meets, emit SplitStep event and state -> SplitHop. In SplitHop, once a directional movement is identified (or directly a lunge), go to Moving. In Moving, when lunge conditions meet, emit Lunge event, state->Lunging. In Lunging, when feet distance falls back to normal, emit Recovery event, state->Recovering. In Recovering, when near base, state->Ready and possibly log a "Cycle complete".

Cooldown and One-At-A-Time Logic: We ensure that when one event is triggered, we suppress any other overlapping event until a sensible time passes. For example, once a Lunge event fires, we likely

don't want to fire another Split or Lunge event for at least e.g. 0.3s, even if some noisy signal appears, because the user is in that phase. We also enforce a **feedback cooldown**: when an error is identified and a coaching cue is given (e.g. "widen your stance"), we won't give another different correction immediately. The system will maintain a timer (say ~5–10 seconds or until the next clear repetition) before it will vocalize or highlight a new correction. This is based on coaching practice that focusing on one thing at a time yields better improvement [4]. Internally, we can queue up issues, but the UI/voice will surface them one by one.

B3. Example: Detecting a Split + Lunge Sequence

Imagine a user does a shadow footwork drill: starting at center, does a split hop, then lunges to the right-front corner, then pushes back to center. The detection might go as:

- Frame 100: Both ankles lift (~2cm upward in image) and knees bend -> **SplitStep event** fired (timestamp ~3.3s). State: SplitHop.
- Frame 110: Ankles land (vertical motion reversed down). We measure that the next motion is moving rightward (right hip, right ankle start moving farther from left) -> State: Moving (we could optionally log "Push-off to right").
- Frame 150: Right foot far from left, right knee angle ~90° -> **Lunge event** fired (timestamp ~5.0s, type "RightFrontLunge"). State: Lunging.
- Frame 180: Right foot begins coming back, distance decreasing. By Frame 220, feet are back near shoulder width and near original center position -> **Recovery event** (timestamp ~7.3s). State: Ready.
- Throughout, our system also logs any form issues, e.g., at the Lunge event, maybe stance width was 1.8× shoulders (which might be fine) and knee angle 90° (fine), but perhaps torso was very tilted. It would tag that. At Recovery, it might note time = 2.3s from lunge to ready (maybe a bit slow).

These events feed into coaching logic (one cue at a time: if the biggest issue was torso lean, that will be the one highlighted first).

B4. Leveraging Existing Patterns or Libraries

The approach above is largely custom, but we can draw from similar projects: for instance, squat or jump detectors often use joint angle thresholds with hysteresis. The literature on gait event detection (for walking/running) also employs threshold + timing rules for heel-strike and toe-off events, achieving high timing accuracy [22]. Those algorithms use angular velocity thresholds with a required quiet period (hysteresis) to avoid double-detection, which is analogous to our approach (e.g., requiring the knees to straighten to a certain extent before another "bend" can count as a new hop). In other words, these rule-based methods have precedent for real-time use due to their reliability and transparency [22].

We might not find an off-the-shelf library specifically for "badminton footwork FSM," but the logic is straightforward to implement. We will keep the rules in a configuration (easy to adjust thresholds over time). If needed, libraries like `transitions` (Python) could manage state, but a simple coded FSM might suffice.

In summary, this rules-first event detection emphasizes precision and debuggability. It ensures we correctly label each phase of footwork, which sets the stage for giving specific, actionable feedback per event (timing of split, depth of lunge, speed of recovery, etc.). As data grows, we could later train an ML model to detect events or classify footwork patterns (some research already classifies footwork types

using YOLO pose [23]), but for V1 the handcrafted approach is more controllable and **avoids the need for a large labeled dataset** upfront.

C. Existing Open-Source Resources to Reuse

A wealth of open-source projects and SDKs can jump-start ShuttleSense development. Below we survey relevant tools in categories: **pose estimation frameworks, smoothing filters, analysis/visualization libraries, sports-specific projects, annotation tools, and AI chat frameworks.** Each is evaluated for readiness, license, and fit for our needs. (See Deliverable 1 for a summary table of 20+ items). Here we highlight particularly useful resources:

- **Pose Estimation Frameworks:**
- *MediaPipe by Google – Provides ready-to-use pose solutions on mobile (Android/iOS) and desktop. We can use MediaPipe's pre-trained models (BlazePose) through a high-level API (with built-in tracking and smoothing). It's cross-platform and Apache 2.0 licensed, making it safe for commercial use. MediaPipe's on-device performance is excellent (real-time camera apps) and it even has a WebAssembly version for in-browser use [24]. For ShuttleSense, MediaPipe could handle real-time tracking in a native mobile app or a web demo without writing custom inference code.* Limitation:* The native graphs can be a black box; customizing the model or extracting intermediate data may require digging into their framework. But overall, a robust starting point for pose detection.
- *TensorFlow.js PoseDetection* – Includes MoveNet and BlazePose models for browser. If we want a purely web app (no server), TF.js allows running MoveNet Lightning in-browser at ~20-30 FPS on a decent laptop (less on phones). It's entirely client-side (great for privacy). We can leverage demos from TensorFlow that combine MoveNet with exercise rep counting [25]. *Limitation:* Browsers have varying speed; older devices will struggle. Also, implementing our complex analysis in JS might be less convenient than Python on a server.

- *OpenMMLab MMPose* – A comprehensive PyTorch toolbox with many pose models (including **RTMPose** for real-time and high-accuracy models). It's **Apache 2.0** [7]. We could use MMPose to experiment with different backbones or fine-tune on any badminton-specific data if needed. It also has utilities for pose tracking across frames. *Limitation:* It's geared towards researchers – fairly heavy and not optimized for mobile deployment. Likely more than we need for V1, but a good reference (and their pretrained weights might be useful if we needed, say, a whole-body model with foot keypoints).

- **Pose Smoothing & Tracking:**

- *OneEuroFilter (GitHub: casiez/OneEuroFilter)* – A reference implementation (C++/Python) of the 1€ filter [6]. We can grab the Python code (it's very short) and integrate it. License is MIT (permissive).
- *FilterPy* – A Python library for Kalman filters and Bayesian filtering. Could be used if we attempt a Kalman smoother for joints. (MIT license).

- *MediaPipe internal filter* – As noted, using MediaPipe's own smoothing (enabled by setting `enable_smoothing=True` in pose options) is an option if we stick with that framework, saving us from implementing a filter.

- **Action/Event Detection Libraries:**

- There isn't a plug-and-play "sports event detection" library that fits badminton directly. But we find inspiration in projects like *RepCount* or *Fitness AI Trainer* which detect exercise phases. For

example, **yakupzengin/fitness-trainer-pose-estimation** on GitHub uses MediaPipe to count squats and give feedback on form [26]. It's MIT licensed. We can see how they structured feedback ("Back straight!" etc.) triggered by angle thresholds. Their focus is static exercises, but conceptually similar (identify an event – a squat – then check angles at bottom position).

- Another example, **giaongo/RepDetect** (Apache-2.0) – an Android app that counts reps and gives real-time feedback using ML Kit Pose. They trained classifiers for pose recognition of specific exercises. This is a bit different from our rule-based approach, but their app design (voice cues, storing exercise history) informs us. Notably, they list "what didn't work": one issue was classification confusion for similar poses [27] – reinforcing our decision to start with simple rules rather than an ML classifier that might confuse footwork patterns.

- If we look at academic work, *Badminton footwork classification* has been attempted with deep learning: Jannet et al. 2024 used YOLOv8 to classify footwork into six corner movements, achieving ~63% mAP [23]. This demonstrates that ML can identify footwork type, but also shows it's non-trivial (63% mAP leaves room for improvement). We prefer to explicitly code corner logic (like "which foot moved, which direction") to get 100% precision on classifying a lunge direction, rather than rely on a model that might misclassify.

- **Multi-View / 3D Pose Tools:** (for future use, but note briefly)

- *OpenCap (Stanford)* – An open-source project for 3D motion capture using **two iPhones** and ArUco markers for calibration [28] [29]. It uses a cloud pipeline: extract 2D keypoints with an algorithm like OpenPose, then optimize a 3D skeleton. They report much higher accuracy with multi-view, especially reducing occlusion issues [29]. License is likely BSD or similar (the code is on GitHub). For ShuttleSense down the road, OpenCap provides a blueprint to incorporate multi-view: sync cameras (they do cross-correlation of audio clap to sync frames [30]), calibrate with known court dimensions or markers, then fuse poses. We will not need this for V1 (single-camera focus), but it's a valuable resource if we add multi-cam support later.

- **Badminton-Specific Projects:**

- *badminton-pose-analysis (Deepak Talwar et al.)* – A 2019 CalHacks project [31] that compared amateur vs pro players' poses for certain shots. They used OpenPose on broadcast footage and even did perspective transforms to a bird's-eye view using court lines [21] [32]. This is clever: by knowing court dimensions, they mapped 2D pose into real-world coordinates to measure lunge distances accurately [33]. ShuttleSense can adopt a simplified version: if the video shows the court and we can identify the court lines, we could calibrate distances (perhaps not V1, but a future enhancement to quantify, say, "lunge was 1.8m"). The code from this project (113⭐, no explicit license noted) shows how to use homography for sports videos.

- *Academic papers:* There are research papers on badminton shot or strategy analysis using pose [34] [35]. One study combined pose with shuttle trajectory for shot recognition [36]. While beyond our immediate scope, these hint that if we ever incorporate shuttle tracking, combining it with pose dynamics yields richer insight (e.g., recognizing a "net shot" vs "smash" based on pose).

- **Visualization & UI Libraries:**

- Drawing the pose skeleton and events on video can be done with OpenCV (for server-side rendered videos) or on the web canvas with simple JS drawing. If using MediaPipe in a browser, they provide a ready canvas output. For a more interactive UI, we might use **three.js** or

Babylon.js to display a stick figure overlay or even a 3D avatar (if we had 3D pose). But for now, a 2D overlay is sufficient.

- There are also specialized tools like **Kinovea** (open-source sports video analysis software) which allows manually tagging and measuring angles. While not directly integrable, it's a reference for the kinds of measurements coaches use.

- **Annotation Tools:** (for building datasets or evaluating our detections)

- **CVAT (Computer Vision Annotation Tool)** and **LabelStudio** – both open source (CVAT is MIT licensed) web apps for labeling video frames or segments. If we decide to create a small ground truth dataset of footwork events (to evaluate our system or train future ML), these tools can let a human annotator mark events in video timelines. We can also use them to verify our automatic detections by overlaying and adjusting. For now, we likely won't invest heavily in custom datasets, but it's good to note these for evaluation.

- **Chatbot and RAG (Retrieval-Augmented Generation) Frameworks:**

- **LangChain** (Apache 2.0) – A popular framework to build LLM-powered applications with retrieval. We can use LangChain to take a question, look up relevant info from the session data (by converting our JSON report to a set of documents or passages), and then let the LLM generate an answer using only that info. LangChain has retriever and memory components to handle this.
- **LlamaIndex (GPT Index)** – Another library that can ingest a JSON or structured data and create an index for question answering. Could be useful to map questions like "How was my footwork?" to the events in JSON.
- **Guardrails AI** (Apache 2.0) – A toolkit to add validation to LLM outputs [37]. We can define "rail" rules, for example: every response must include a reference timestamp if it makes a claim about the session, and it should refrain from giving advice not backed by data. This can help enforce the no-hallucination requirement. Guardrails can be integrated with LangChain or used standalone in the generation step.
- We will likely use OpenAI's GPT-4 or similar as the chat model initially (not open-source, but via API). If open-source LLMs catch up in capability by 2026, we could consider hosting one fine-tuned on coaching style. Regardless, the grounding via retrieval is what ensures correctness.

In summary, we have plenty of building blocks. **Table 1** (in Deliverables) will enumerate the key tools and libraries, each with how it fits into ShuttleSense. By leveraging these, we avoid reinventing the wheel – focusing our effort on the glue: integrating pose tracking with domain-specific rules and a great user experience.

D. Similar "AI Coach" Products & Prior Art

Before finalizing our design, it's insightful to learn from existing AI coaching and fitness products. Several systems in the market or research provide real-time feedback on human movement. We examine what works well and common pitfalls:

- **Fitness Form Apps (e.g. mirror-based trainers, home workout AI):** Many apps use pose estimation to count reps and check form for exercises like squats, push-ups, yoga poses (examples: *Kaia Health*, *Tempo Move*, *Onyx*). Successful elements include: immediate visual feedback (skeleton overlay, or green/red outlines when form is correct/incorrect) and simple cues ("Knees behind toes!", "Straighten your back"). They often use **single-cue feedback**, exactly as we plan, because users can only focus on one correction at a time. They also tend to be

forgiving – not every wobble triggers an alert, to avoid annoying the user with constant "bing-bing" feedback. Instead, they choose significant deviations or consistent errors to comment on. We will emulate this tolerance via our cooldown and by requiring an error to persist for a short time or multiple reps before flagging.

- *What fails:* Overly generic or constant nagging. If an app says "Work on your form!" without specifics or keeps repeating "Incorrect! Incorrect!" on every rep, users get frustrated. We have seen user reviews complaining about apps that misjudge due to poor detection (e.g., calling a correct rep wrong because of angle issues) – this erodes trust. Thus, precision in detection and phrasing feedback constructively ("Try widening your stance a bit on lunges for better balance, e.g. at 00:45 [5] ") will differentiate ShuttleSense.

- **Sports Technique Analytics:**

- *HomeCourt (Basketball)* – This app (by NEX Team) is known for analyzing basketball shots using a phone camera (it detects shot angle, release time, makes/misses). It provides instant stats and video replays with overlays. The key takeaway is the gamification and **data-driven** coaching: e.g., "You made 7/10 from the corner, release was 0.5s on average." ShuttleSense can similarly present numbers (how many lunges, average recovery time, etc.) to give a quantitative sense of progress. HomeCourt also got traction by being easy to use (just prop your phone, no wearables) – reinforcing our aim for a single camera, automated approach.

- *Tennis/Golf analysis apps:* There are products like *SwingVision* (for tennis) or *Zepp Golf* that analyze swing technique through video. They tend to focus on very specific metrics (serve speed, swing plane) and rely on either high-speed cameras or sensors. In badminton, one prior example is Sony's Smart Tennis Sensor, but that's a racket sensor focusing on shot speed, not footwork. It means our focus on footwork is relatively novel – existing badminton tech coaches (like there's a startup *Clutch* mentioned in badminton circles) focus more on match stats and shuttle placement. *Clutch* (for racket sports, originally padel) sets up permanent cameras on court to track players and shuttle in real time [38] . It produces **automated highlight reels and analytics** (distance covered, shot stats). While Clutch's scope is broader, one relevant feature is it gives players a *"Clutch Score"* and improvement tips post-match. They have an AI coach element but likely not as detailed on footwork technique for each rally. Nevertheless, one thing to learn is their **highlight videos with overlays** – for ShuttleSense, a compelling report could include short clips of the user's "best rally" or "worst stance" for self-review (with skeleton overlay). That might be beyond V1, but it's an engaging way to present analysis.

- **Physical Therapy & Rehab Coaches:**

- Systems like *SWORD Health* or *Kaia Back* use pose tracking to ensure patients do exercises correctly. They often operate in a guided session mode (the app says "Do 10 lunges", monitors you, then scores your performance). They succeed by being encouraging and clear ("Great job!" when you do it right, gentle corrections when not). We should incorporate positive feedback too, not just errors – e.g., identify a *good* rep or improvement ("Nice wide stance on that last lunge – much more stable."). This keeps motivation up.

- They also emphasize **safety and personalization** – e.g., adjusting difficulty if a user has limitations. For ShuttleSense, personalization might mean adjusting thresholds as we learn the user's typical ranges (for instance, if someone is very short, their absolute stance width might always be small – instead of blanket "too narrow", base it on their normal). Over time, we can adapt these thresholds per user baseline (see Deliverable 4 on calibration).

- **Prior Academic Prototypes:**

- There was a Microsoft Research project "CoachAI" that gave feedback on basketball defensive stance using a Kinect. It found that giving one suggestion at a time and using simple language was key for user acceptance.
- Another example: **"PerfectSquatChallenge"** – an AI demo by Google using MoveNet to score your squat form. They applied a scoring system and had a competitive angle (points for each well-done squat). For badminton footwork, a scoring (like 8/10 footwork score for the session) could be motivating. We can derive such a score from metrics (e.g., average recovery time relative to a benchmark, proportion of splits done on time, etc.).

Common Pitfalls and How We Avoid Them:
- **False Feedback from Tracking Errors:** Many AI coaches struggle when the pose estimation fails – e.g., if the model loses a limb, it might say "bring your arm down" erroneously. We mitigate this by using model confidence: if confidence is low or keypoints are not visible, we prefer to say nothing than say something wrong. Also, by smoothing and filtering, we reduce the chance of transient blips causing a feedback trigger.
- **Latency:** If feedback comes even 1–2 seconds late in a real-time scenario, it's disorienting. That's why we choose fast models and minimal processing. For live mode, the pipeline should ideally stay under ~100ms end-to-end (10 Hz). If it can't, we might restrict live feedback to simpler cues (like beep when a split is detected late) rather than detailed commentary on the fly. The post-session analysis can be more thorough since it's not real-time.
- **User Experience:** Some products require complex setup (calibrating cameras, wearing markers). We commit to simplicity: just set up your phone on a tripod at roughly recommended angle (we'll provide guidance), hit record or live, and go. If something is not ideal (camera angle), our system will still attempt to work and just note any uncertainties ("Camera angle made it hard to see left foot – some feedback may be missing."). User should always get *something* useful without needing to be a technologist.

In summary, existing AI coaches validate that the technology can be engaging and helpful, but only if it's accurate and user-friendly. ShuttleSense will stand on the shoulders of these prior efforts, adopting their best practices (real-time speed, clear specific feedback, gamified metrics) while avoiding their missteps (false alarms, information overload, privacy insensitivity).

E. Minimal Web App Architecture (Personal-Use Friendly)

ShuttleSense's architecture will prioritize **simplicity, cost-effectiveness, and privacy**, aligning with a personal web app use case. We outline two modes: (1) **Video Upload Analysis** (primary focus, not real-time) and (2) a prototype **Real-Time mode** (for live practice with immediate audio cues, likely as a mobile app later on).

E1. Video Uploader & Cloud Processing Pipeline

User Flow: The user records their session with their phone (or any camera) and uploads the video via the ShuttleSense web app. After a short processing delay, they receive an interactive report of their footwork analysis and can enter a chat interface to ask questions about the session.

Front-End: A web application (could be a simple React app) that handles:
- Video upload (to cloud storage or directly to the backend).
- Displaying a processing status/progress.
- Once ready, displaying the analysis report: possibly a timeline with events marked, summary stats, and

key feedback points. We can overlay skeleton stick figures on the video or show snapshot images at certain moments (since by default we avoid storing full video on server, we might rely on the user's browser to display the original video file synced with an event timeline).
- A chat UI (text input + conversation display) where the user can query their "AI coach" about the session.

Since privacy is a priority, we **do not permanently store the raw video** on the server. The video might be uploaded to a temp location (e.g. a signed URL to Google Cloud Storage) for processing, but after processing, we delete it unless the user opts to keep it. The key outputs (pose keypoints, events, metrics) are stored in a JSON (which is tiny relative to video) and possibly some extracted image clips for evidence. The front-end can locally cache the video if needed for replay with overlay (or simply ask the user to keep the video file if they want to watch alongside the analysis).

Back-End Processing:
- We will use a serverless container (Google Cloud Run) or a small VM that wakes on demand. The backend is essentially a **Python service** with the pose model and analysis code. When a video is received (or available via cloud storage link), it runs the pipeline:
1. **Decode Video Frames:** Using OpenCV or FFmpeg, read frames at a suitable frame rate (e.g. 30fps). If the video is higher FPS (60), we could downsample if not needed, but for accuracy of fast actions maybe keep it if performance allows. We'll also handle resizing frames to the model's input size (e.g. 256x256 or 256x BlazePose).
2. **Pose Inference:** For each frame, run the pose model (MoveNet/BlazePose). On CPU this would be slow, so we'd utilize a GPU if possible. Cloud Run now supports GPU accelerators in some regions, or we could use Vertex AI Batch for more heavy compute. Since cost is a factor, we might opt for Cloud Run with CPU if it's tolerable: MoveNet Lightning on CPU might be borderline for 18k frames (perhaps ~0.02s/frame => ~6 minutes). With a T4 GPU, we could cut that to under 1 minute easily. We can also do smarter frame-skipping for analysis: e.g., analyze at 30fps but only record events at necessary fidelity (maybe we don't need every single frame's output, just enough to catch events).
3. **Pose Smoothing:** As frames are processed, we feed keypoints into our smoothing filters. This can be done in an online manner (filter as we go) to avoid storing huge arrays. By end, we have a timeline of smoothed pose data.
4. **Event Detection:** Run our rule-based event detector over the pose sequence (this could be in the same pass as above or a second pass). It will output a list of events with timestamps and any attributes (e.g., "Lunge right, stance_width=1.4x, knee_angle=95°").
5. **Metric Calculations:** Compute session-level metrics: e.g., average recovery time, count of lunges, mean stance width, etc. Also identify "mistakes" or anomalies: e.g., "3 out of 5 lunges had stance width <0.8x (too narrow)".
6. **Compose JSON Report:** We create a structured JSON (or possibly a more domain-specific format) containing all relevant data. For example:

```
{ "events": [
     { "type": "SplitStep", "ts": 3.3, "notes": {"timing": "late"} },
     { "type": "Lunge", "ts": 5.0, "direction": "ForehandFront",
       "metrics": { "stance_width_ratio": 1.1, "knee_angle": 90,
"torso_lean": 25 },
       "issues": ["stance_narrow"]
     }, …
   ],
   "summary": { "total_lunges": 5, "avg_recovery_time": 1.4, "issues":
```

```
["stance_narrow: 3/5 lunges", "slow_recovery: 2 instances"] }
}
```

(We will refine this schema in Deliverable 3, but essentially all the analysis results are here.)
7. **Store/Return Results:** Save the JSON (e.g., in a Cloud Storage bucket or database) keyed to the user and session. This JSON is what the front-end will use to display the report and also what the chat will query. We do **not** store the video by default – so privacy risk is minimized (the JSON by itself has no images of the user, just coordinates and derived metrics).
8. **Optional - Generate visual assets:** We could create some images or clips to embed in the report, such as a composite image of the user's skeleton at deepest lunge vs ideal skeleton (if we had an ideal reference). Or a chart of their movement over time. These could be generated now or on-the-fly in front-end using the data. To start, likely we stick to lightweight visuals (like drawing skeleton on canvas using the JSON points).

Performance & Cost: A 10-minute video (600 seconds). At 30fps, ~18k frames. If using a GPU instance (like an NVIDIA T4), MoveNet or BlazePose can process easily >100fps [9], so roughly ~3 minutes compute time. Cloud Run charges per use – say it takes 180 seconds on a GPU ($1–2/hour) => $0.05–$0.10 per video. Additional overhead for data transfer and storage is negligible (JSON maybe 1MB). If usage is low volume (personal use or small scale beta), this is very affordable especially with credits. On CPU only, 18k frames * ~10ms = 180s, plus overhead – could be ~4–5 minutes, which might be acceptable if user is willing to wait a bit. We'll likely start with CPU to keep infrastructure simple, optimizing only if needed.

Scalability: If this were to serve many users, we'd consider a more robust setup (e.g., a queue of video processing jobs, auto-scaling workers, etc.). But for personal or small pilot, a single Cloud Run instance processing one video at a time is fine.

E2. Real-Time Feedback Mode (Future Mobile App)

Though the question focuses on video analysis, we keep in mind real-time use (e.g., training alone on court with your phone giving live cues). This likely means an **on-device** solution to avoid network latency. Here's how it could look:

- A mobile app (Android first, perhaps) running MediaPipe or TF Lite MoveNet locally. The camera feed goes through the pose model in real-time. We apply our smoothing and event logic in a streaming fashion. When an event or mistake is detected, the app gives an **audio cue** (e.g., text-to-speech: "Widen your stance." or a simpler sound if that's too advanced). The app could also display an overlay (like a ghost skeleton or highlight) on the phone screen, but in practice the athlete might not look at the screen while moving – so audio or haptic feedback is preferable.
- Latency here needs to be minimal – ideally under 200ms from event to cue. Our pipeline (MoveNet on phone ~30ms/frame, plus logic negligible) should meet that.
- We'd implement a similar state machine, but possibly a bit simplified to reduce chatter. For example, in live mode we might only focus on one type of feedback per session (user chooses "work on my split step timing" and the app then only says if you were late or not, rather than many kinds of corrections). This avoids confusion.
- The app can record the session concurrently (storing video or the pose data) so that the user can later upload for a detailed analysis. This way the real-time feedback addresses immediate cues, and the post analysis covers the deeper dive.

For now, the real-time mode is a "Phase 2" after we nail the analysis pipeline. But making sure our core components (pose estimation, filtering, event detection) are portable to run on device (via TF Lite, etc.)

will ensure we can transition to this mode. The **Recommended V1 Stack** (Deliverable 2) will outline choices that favor cross-platform deploy (e.g., choosing a model that has a TF Lite or CoreML version available, which MoveNet and BlazePose both do, and writing detection logic in a device-friendly language if needed).

E3. Cloud vs Vertex AI Considerations

Google Vertex AI could host models or run batch jobs. For example, we could deploy the pose model as a Vertex Prediction endpoint. However, that introduces overhead (each frame would be an API call unless we batch frames). It's simpler and likely cheaper to run our own inference inside a container given the known model. Vertex is more useful if we needed to train models or use AutoML, which isn't our focus for V1. Given our budget-conscious approach, we'll stick to Cloud Run or possibly Cloud Functions (though video processing is a bit heavy for a short-lived function, better to use a container).

Storage & Security: All user data (video, pose JSON) will be in that user's own space (if multi-user, we segregate by user auth). We'll use expiring URLs or require login to view results. Since the plan is personal use (maybe even open-source the tool for users to run locally), security is not a huge issue beyond standard cloud practices.

Summary of Arch (V1):
- **Frontend:** Web app (HTML/JS) for upload, results, chat – possibly can be a static site + minimal logic.
- **Backend:** Python on Cloud Run: does pose & analysis, returns JSON. Possibly separate endpoint for the chat (or we do chat directly in frontend calling OpenAI API with the JSON context – could do that to avoid storing any chat data on our server).
- **Data:** Pose/Events JSON (small, can be stored in Firestore or just returned directly). Raw videos not stored (or stored temporarily in a secure bucket).
- **Processing time:** ~minutes, user is okay to wait and then get result. They can review it at leisure.

This architecture is lightweight, leveraging the cloud only when needed and keeping costs low. It also aligns with privacy – the persistent data is abstract (poses), and if a user deletes their session, it's just deleting some JSON (which is meaningless outside context).

F. Multi-View & Future Extensions (Beyond V1)

Multi-view analysis (using two or more cameras) is a powerful enhancement for the future, but not part of our V1 scope. We briefly consider it to future-proof our design:

- **Why Multi-View:** With two camera angles (e.g., one behind, one side), we can perform 3D pose triangulation [29]. This would give actual distances (in meters), more accurate stance width, jump height, and the ability to detect things like whether a foot was actually off the ground or just occluded. It also helps when one camera loses sight of a limb (the other angle might see it). Overall, multi-view would greatly improve accuracy [29].

- **Challenges:** Needing two devices filming synchronously introduces complexity. The feeds must be synchronized (within, say, 10ms). Solutions include clapping at start (to sync audio spikes) [30] or using network time if devices are connected. Calibration is also needed: we'd have to calibrate both cameras to a common coordinate frame. OpenCap uses a one-time calibration with markers and known distances on the ground [28] – in badminton, the court lines could serve as calibration markers since their dimensions are known (e.g., the service line, baseline). We could

ask the user to position cameras and perhaps do a quick calibration move (walking around). This is probably too cumbersome for average users, but possible for advanced ones or coaches.

- **Open-Source for Multi-View:** As mentioned, **OpenCap** provides algorithms and perhaps code for multi-view calibration and 3D reconstruction [28]. Also, OpenPose has a 3D demo if given calibrated multi-camera setup. We can draw on those when we get there.

Given the above, our architecture in Section E keeps multi-view in mind by not hard-coding assumptions that there's only one camera forever – e.g., our data format could allow multiple "tracks" of pose, and our analysis could fuse them if present. But for now, **we deliver value with a single camera**. The user story is simpler (just use your phone), and we avoid an "accuracy trap" of chasing perfect 3D data too early. Many single-camera products (e.g., the ones mentioned in D) have shown you can extract meaningful coaching tips in 2D. We will join that league first, and treat multi-view and possibly **shuttlecock tracking** (another advanced feature) as part of the long-term roadmap once the foundation is solid.

Now, we proceed with the concrete deliverables, summarizing our findings and decisions:

Deliverables

1. Landscape Table of Relevant Technologies (20+ Items)

Below is a table surveying libraries, models, and tools that can be utilized for ShuttleSense, with notes on readiness, license, usefulness, and limitations:

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
MediaPipe Pose (BlazePose) [19] [8]	Full-body 33-point pose estimator by Google. Runs real-time on mobile (Android/iOS) and web (WASM). Offers tracking and smoothing out-of-box.	Yes – used in many apps	Apache 2.0	**Fast, on-device** pose with foot keypoints; cross-platform SDK (no ML expertise needed to integrate). Proven in fitness and AR apps.	Limited multi-person capability. Some jitter/errors on very fast motions or occlusions.

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
MoveNet (Lightning/ Thunder) [1]	Ultra-fast 17-point pose model (TF Hub). Lightning ~50 FPS on phones [1], Thunder is bigger for accuracy. Available in TFLite, TF.js.	Yes – production ready	Apache 2.0 (model under TensorFlow license)	**High FPS, low latency** – great for real-time drills. Simple model, easy to use (TF or TF.js). Good accuracy on common poses.	Only 17 keypoints (no toes). May lose tracking during complex flips or if multiple people present (designed for single-person).
YOLOv8-Pose (Ultralytics) [9]	One-stage detector + pose model (COCO keypoints). Very compact (Nano ~4M params) and fast (60+ FPS on GPU [9]). Can detect multi-person and objects concurrently.	Yes (code & models released)	GPL-3 (code) Model weights: permissive	**Unified solution** – can detect the player and keypoints in one go. Good accuracy (used in sports research [10]). Scalable (bigger models for more accuracy).	GPL license means integrating the code is tricky for commercial use (workaround: use exported model). Lacks foot-specific kpts. Not as tried-and-true on mobile as BlazePose/ MoveNet.
OpenPose (CMU)	The original multi-person pose estimator (body, hands, face, feet). Highly accurate on full body keypoints. C++ implementation with Python API.	No (real-time only on high-end PC GPUs)	Non-commercial (custom lic) [12]	Provides *feet keypoints* and very detailed pose. Could be used offline to validate other models or augment dataset labels.	**Very slow** on CPU/mobile. License forbids commercial use without fee [12]. Large model (~200MB), high GPU memory usage.
AlphaPose (MVIG SJTU)	Accurate pose estimator (can do whole-body with feet, hands). PyTorch based. Achieved SOTA accuracy on COCO.	Partial (realtime on GPU, not mobile)	Non-commercial (custom lic) [14]	High accuracy, tracking multiple people with IDs. Could be used for generating ground-truth on videos (for research/ validation).	Non-commercial license. Heavy computation (not edge-friendly). Overkill for single-person scenario.

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
OpenMMLab MMPose [7]	Comprehensive toolbox supporting many models (HRNet, ViTPose, RTMPose, etc.) for 2D/3D pose. Training & inference pipelines.	Yes (for dev use)	Apache 2.0	Offers **ready models** and training code. Notably RTMPose is a real-time model (90+ FPS on GPU with ~75 COCO mAP [39]). We could utilize their pretrained models or benchmark various architectures easily.	Complex to integrate for a quick app – meant for ML engineers. No built-in mobile deployment (would have to export).
Google ML Kit: Pose Detection [16] [17]	Google's on-device ML SDK (Android/iOS). Pose API uses BlazePose under the hood (offers "accurate" and "fast" modes).	Yes (production)	Proprietary (free to use)	Easiest way to get pose in native apps – just call the API. Handles camera, permissions, etc. **Good for our potential mobile app** real-time mode.	Not open-source. Slightly behind latest models (but continually improving). Customizing model not possible.
OneEuro Filter [6]	Simple filtering algorithm for noisy signals with minimal lag. Available implementations in multiple languages.	Yes	MIT (reference code)	**Real-time smoothing** for pose points to reduce jitter [40]. Very lightweight – great for embedding in app or web.	Needs tuning per signal. Might not fully remove jitter if movements are extremely rapid (could consider higher-order filters for that).

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
Kalman Filters (e.g. FilterPy)	Libraries for Kalman filter implementation. Can be used to stabilize time-series and predict occluded points.	Yes	MIT	Could improve stability when parts go occluded – e.g., keep predicting foot location for a few frames. Useful in **smoothing pose trajectories** beyond low-pass filtering.	Requires a motion model; if mis-tuned, can lag or overshoot. More complex than one-euro for marginal gain in our scenario.
transitions (FSM library)	A lightweight Python finite-state machine library for implementing stateful logic.	Yes (mature)	MIT	Can help structure the event detection state machine in a clean way (define states, transitions with conditions). Useful for clarity and maintainability of our rule-based system.	Not strictly needed – our logic is simple enough to hardcode. But could be handy if we add more states or modes.
Yakup's Fitness AI Trainer [26]	Open-source project using MediaPipe to count exercise reps and give form feedback (squat, push-up, curl). Web frontend + Flask backend.	Demo-level (works, not a product)	MIT	Provides sample code for integrating pose detection and giving feedback. We can reuse ideas for UI (e.g., overlay skeleton on video, how to prompt user). Also shows how to do basic form-checks (angle thresholds) in practice.	Narrow scope (few exercises). Not focused on footwork or leg movement in space. But logic for rep counting could inspire our event counting.

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
RepDetect (Android) 	Student-built Android app that counts reps and classifies exercises in real-time using ML Kit Pose. Has voice cues and stores performance history.	Demo app (open source)	Apache 2.0	Great **mobile blueprint**: how to run pose in real-time and give audio feedback. Also shows a data pipeline: they trained a classifier on poses – could inspire future ML in ShuttleSense for classifying footwork style.	Focused on gym exercises; their classification approach had issues with similar poses [27] (reinforces that careful feature engineering or more data is needed for classification – hence our rule-based first approach).
Deepak's Badminton Pose Analysis [31] [32]	Hackathon project analyzing pro vs amateur pose for specific badminton shots. Used OpenPose to extract pose and homography to normalize camera perspective [21].	Prototype (2019)	(No license given, assume non-commercial)	**Badminton-specific insight**: how to measure lunge distance and consistency by referencing an expert model. We might use similar approach to compare user to an "ideal" (if we gather reference metrics from pros). Homography use to get real distances is a big plus for future accuracy [33].	Limited to single frames of specific shots (not continuous footwork). Depended on broadcast video (side view) – may not directly translate to user tripod view. Code is not a plug-and-play library, more of a custom script.

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
OpenCap (Stanford) [28]	Open-source system for **3D motion capture from video**. Uses multiple smartphone videos + calibration to compute 3D kinematics and even joint forces.	Research (in use by biomechanists)	MIT/BSD (open-source)	**High-accuracy 3D**: If we ever incorporate dual-camera, OpenCap provides algorithms for sync, calibration, and 3D reconstruction [29]. It's essentially a pipeline we could adapt to get 3D pose for badminton.	Needs two cameras and markers – not at all for V1 casual use. It's cloud-based (they have their own server pipeline) – we'd likely just learn from it rather than integrate directly now.
COCO & Sports Datasets (COCO, PoseTrack, etc.)	Large annotated datasets for pose estimation and tracking. COCO has 2D poses; others like PoseTrack have videos. Also domain-specific ones e.g., badminton pose dataset by YOLOv8 paper (4k Kinect images) [41].	Yes (available for training)	Varies (COCO is CC BY 4.0)	If we need to retrain or fine-tune models (not in V1 scope), these are the go-to data. For instance, a badminton-specific pose dataset [41] could improve pose accuracy on badminton stances (the YOLOv8-ELA model was trained on such data for better results [42]).	Training models is outside V1. We will rely on pre-trained. But knowledge of these helps if we face pose estimation failures on uncommon badminton poses (like a deep crouch defense).

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
CVAT / LabelStudio (Annotation)	Web-based tools to label video frames, keypoints, or action segments.	Yes	Apache 2.0 (CVAT), Apache 2.0 (LabelStudio)	Useful for **evaluating and improving** ShuttleSense. We can have a human (or the coach user themself) label correct/ incorrect instances in a video, and compare with our detection. Also, if we develop an ML model (say to classify “late split”), we’d need labeled examples – these tools assist in creating that data.	Manual annotation is time-consuming. Probably used internally for validation rather than user-facing.

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
LangChain (for Chat)	Framework for building LLM apps with retrieval. Simplifies connecting a knowledge base (our session data) to the LLM via prompts.	Yes (widely used)	MIT	Key for our **Coach Chat**: we can convert our JSON analysis into a set of text pieces (e.g., “At 5.0s: Lunge, stance 0.8x, too narrow”) and have LangChain’s retriever find relevant pieces for a user’s question (“How was my stance?” -> finds all stance-related notes). It helps ensure the LLM has the right info to formulate answers.	Another dependency to manage, but lightweight. Need to carefully engineer prompts so that the LLM doesn’t stray from provided data.
Guardrails AI (Shreya’s) [37]	Toolkit to enforce constraints on LLM outputs (like must include citations, or must not mention unapproved content).	Yes (dev community)	Apache 2.0	Adds a **safety and correctness layer** for our chat. We can define that any factual claim must reference a timestamp from the JSON, and Guardrails can validate the LLM output against that. This greatly reduces chance of hallucinations or inappropriate advice.	Still a developing field; may need writing schema/ rules. But since our domain is narrow, rules can be straightforward.

Name & Link	What it Does	Ready for Prod?	License	Usefulness to ShuttleSense	Limitations
OpenAI GPT-4 (or GPT-3.5)	The language model we'd likely use for generating chat responses (via API).	Yes (via API)	Proprietary (pay per use)	**Best-in-class understanding** to interpret user questions and compose friendly, coach-like answers, given our data. We can instruct it to be a badminton coach persona, using the retrieved stats to answer.	Cost per message (but small scale personal use is fine with a few cents per chat). Also must guard against it going outside bounds – solved by grounding and guardrails as above.

(References in table where applicable: performance claims, licenses, etc.)

2. Recommended V1 Stack (Single-Angle Video)

Pose Estimation: Use **MediaPipe BlazePose (Full)** on the server for video analysis due to its accuracy and foot keypoints, and **MoveNet Lightning** on-device for any live feedback prototypes. This combo covers both needs. Specifically, we'll run the MediaPipe Pose model via the MediaPipe Python package or TF Lite in Python. It will run ~30 FPS on a decent CPU, faster on GPU. The unified 33 keypoint format gives us rich data (we'll primarily use the subset of 17 keypoints that overlap with MoveNet's schema for consistency, but keep the extra foot points for possibly better stance and jump detection). For real-time mobile, ML Kit's Pose (which is BlazePose underhood) can be used for simplicity – that ensures consistency with our analysis model.

Smoothing: Implement a **OneEuro filter** on the pose keypoints. We'll tune filter params as: min_cutoff ~1.0 (fairly aggressive smoothing) and β ~0.005 for slow-moving joints (to smooth jitter when holding stance), but higher β for feet, e.g. 0.1 to allow quick changes. These values will be experimented with on sample data. Additionally, use a short moving average (window 3-5 frames) on key metrics like stance distance to smooth any remaining jitter. MediaPipe's internal smoothing will be enabled too, so our filter is an extra layer. The result should be a very stable pose time-series with only ~1 frame of latency added.

Event Detection: Use a **Rule-based Finite State Machine** approach as described in section B. We'll code this in Python for the cloud pipeline. Each frame's smoothed pose updates the FSM state and possibly triggers events. For clarity and extensibility, we might define the rules in a config (e.g., JSON or Python dict) so we can adjust thresholds without digging through code. E.g.:

```
rules = {
 "SplitStep": {"kneebend_deg": 15, "ankle_y_diff": -0.02, "within": 0.2},
 "Lunge": {"min_foot_dist": 1.3, "max_knee_angle": 120},
```

```
  "Recovery": {"foot_dist_below": 1.2, "return_within": 2.0}
}
```

These would be interpreted by the FSM logic. One cue at a time will be handled by a simple global flag or timer that suppresses new “issue events” for a few seconds after one is raised. The FSM also ensures sequential logic (split -> lunge -> recovery).

UI Overlay: For post-analysis, the plan is to overlay skeletons and event markers on the video. We will do this by rendering via the front-end: the JSON has all poses (normalized or in pixel coords). We can draw lines between joints on an HTML canvas on top of the video element. For event markers, we could place an icon or highlight at the frame where it happens (or a timeline scrubber). Also, generating a separate “summary video” is optional: we could use OpenCV to draw poses on frames and encode a short clip highlighting key moments. But initially, a simple interactive timeline with the original video is easier (no heavy video re-encoding). The UI might list events like a playlist – click “Lunge at 5.0s” and it jumps the video to that time, with skeleton drawn. This way, the user can see themselves and the AI overlay.

Technology choices: Web app (React or simple JS) for UI, since we need to handle video playback. Backend in Python (FastAPI or Flask on Cloud Run) because of rich ML and CV support. We’ll containerize with all dependencies (MediaPipe, OpenCV, oneEuro code). We ensure the container can use CPU efficiently; if needed, build with GPU support for MediaPipe (though might stick to CPU for simplicity first).

By choosing MediaPipe/MoveNet, we ensure cross-platform: if later we want the analysis entirely on-device, we have equivalent models. Using Python for analysis backend is fine due to the relatively low volume, but if scaling or speed become an issue, we could rewrite critical parts in TensorFlow/C++ or use Node.js with TF.js on server – unlikely needed though.

Summary of V1 Stack: - **Platform:** Cloud-run Python service + Static JS frontend. - **Pose Model:** MediaPipe BlazePose (Full, 33-keypoint) via Python (TF Lite). - **Filtering:** OneEuro (Python code). - **Event Detection:** Custom Python FSM logic. - **Data store:** JSON in memory (or Cloud Storage), minimal DB usage. - **Feedback Output:** JSON report (used for both visual report and chat). Possibly pre-generate some graphs (could be done client-side too). - **Deployment:** Docker container (with mediapipe, etc.), triggered by an upload event or HTTP request. - **Mobile Live Prototype:** Android app using MLKit Pose and the same rules (rewritten in Kotlin), not in initial deliverable but planned.

This stack emphasizes using **existing, well-optimized components** (Google’s pose estimation) and adding our domain logic on top, rather than training new models. It is low-risk and allows quick iteration on the feedback logic which is the differentiator for ShuttleSense.

3. “Coach Report” JSON Specification (and Example)

We design a JSON schema for the session analysis output, which will be used for both displaying the report and serving the chat’s knowledge base. Key sections of the JSON: **events timeline**, **per-event details**, **aggregated metrics**, and **overall summary/comments**. We also include the needed data to reference evidence (like specific timestamp or frame indices).

Schema Outline:

```
{
  "session_info": {
    "date": "2026-01-04",
    "player_name": "User1",
    "video_filename": "session1.mp4",
    "camera_angle": "rear"
  },
  "events": [
    {
      "id": 1,
      "type": "SplitStep",
      "timestamp": 3.2,          # seconds in video
      "duration": 0.15,          # how long (if applicable)
      "properties": {
        "timing": "late",       # could be "early/on-time/late" relative to
cue if known
        "height": 0.10          # approx jump height in meters or relative
(if calc possible)
      },
      "issues": []              # any problems noted here
    },
    {
      "id": 2,
      "type": "Lunge",
      "timestamp": 4.7,
      "direction": "forehand_front",    # which corner, if classified
      "properties": {
        "stance_width_ratio": 0.75,     # relative to shoulder width
        "knee_angle_deg": 85,           # front knee bend
        "torso_lean_deg": 40            # torso angle from vertical
      },
      "issues": ["stance_narrow", "excess_torso_lean"]
    },
    {
      "id": 3,
      "type": "Recovery",
      "timestamp": 6.0,
      "properties": {
        "recovery_time": 1.3    # time from lunge to back to base
      },
      "issues": ["slow_recovery"]
    }
    // ... more events in sequence
  ],
  "metrics": {
    "total_lunges": 5,
    "average_recovery_time": 1.1,
    "average_stance_width_ratio": 0.88,
    "split_hops_count": 5,
    "timely_splits": 2,
```

```
    "late_splits": 3
    // possibly distribution stats, etc.
  },
  "overall_issues": [
    { "issue": "stance_narrow", "count": 3,
      "examples": [ {"event_id":2, "timestamp":4.7}, {"event_id":
5,"timestamp":30.2}, ... ],
      "advice": "Try placing your feet about shoulder-width apart when you
land to improve stability."
    },
    { "issue": "slow_recovery", "count": 2,
      "examples": [ {"event_id":3,"timestamp":6.0}, {"event_id":
6,"timestamp":32.5} ],
      "advice": "Work on pushing off faster after each shot to cut down
recovery time."
    }
  ],
  "summary": "Your footwork session had 5 lunges. Main areas to improve:
stance width (3 instances too narrow, e.g. at 4.7s) and recovery speed (took
>1s twice, e.g. at 6.0s). Split steps: 2/5 were timed well, 3 a bit late.
Keep an eye on maintaining balance (torso lean was high on deep lunges).
Strength: consistency in performing split step before moving each time -
great job on that fundamental!"
}
```

Explanation: - `session_info`: metadata. `camera_angle` might be used to adjust interpretation (e.g., if side view, stance width ratio might be less reliable). - `events`: array of chronological events. Each event has an `id` (so that references can point to it), a `type` (from a controlled vocabulary: SplitStep, Lunge, etc.), a timestamp (in seconds for precision, or could be frame index), and any relevant `properties`. If an event had multiple phases (like a long rally), we might have a `duration`. For lunge, we add a `direction` to know which corner (which can help user picture which one, especially if multiple in session). - Each event also can list `issues` – keys that indicate something was wrong or noteworthy in that event. E.g., stance_narrow, late_split, off_balance, etc. - `metrics`: aggregate numbers for the session. These are useful to show progress over time or compare to benchmarks. E.g. how many lunges, average metrics. We include split timing counts, etc. - `overall_issues`: this synthesizes the repeated issues. Each entry describes one type of mistake or area, how many times it occurred, example event references, and perhaps a generic coaching advice string for that issue. (This is basically a lightweight knowledge base the chat or report can use). We might tag severity or priority as well (if many occurrences or critical issue). - `summary`: a human-readable paragraph summarizing the session. This could be generated by some template or even by an LLM constrained to facts. It's what might appear at top of a report as the coach's overall comment. The chat can elaborate further, but this summary ensures the user who doesn't chat still gets a narrative interpretation of the data. It references specific timestamps for evidence (as we must ground feedback in evidence) – e.g., "(e.g. at 4.7s)" provides a concrete example [5].

We will keep the JSON **machine-readable** (for the chat bot to parse) but also easily convertible to a user report. The chat system will use primarily the structured data (it might not use the freeform summary at all, or use it as a fallback).

Example Scenario (from JSON): In the example above, at event 2 (Lunge at 4.7s), issues "stance_narrow" and "excess_torso_lean" were flagged. This contributed to overall_issues stance_narrow count. The summary mentioned "e.g. at 4.7s" – which a chat could elaborate: "At **4.7s**, during your forehand front-court lunge, your feet were only 0.75× your shoulder width apart [5]. This narrow stance may reduce your stability. Try to land closer to 1.0× (shoulder width)." Here the citation would be pointing to the session JSON (which in the UI can map to the video moment).

Storage format: Likely JSON as shown. If we needed more efficiency or schema enforcement, we could use an actual database with fields, but JSON is flexible and fine for our scale. We just need to ensure the references (event IDs) align, etc.

This spec might evolve with usage, but it captures the needed elements: timeline of events, metrics, and identified issues with evidence. Everything the coach AI will say should be derivable from this data structure.

4. Rule Threshold Starter Pack (Initial Ranges & Calibration)

We initialize detection thresholds based on a combination of coaching literature, our intuition, and a little buffer for model noise. These will likely be tuned per user eventually, but here are starting values and how to adjust:

- **Split Step Timing Window:** *Definition:* If the user had an opponent, "on-time" means landing the split as opponent hits. In solo drills, we define on-time relative to a cue or movement start. For now, if a split hop occurs within **±0.2 seconds** of a nominal cue (or just before a push-off), we call it "on-time". If later than that (e.g., user already started moving feet before finishing split), mark "late". **Threshold = 0.2s window**. Calibration: If we have multiple splits in a session, we can see their distribution relative to moves and tighten or relax accordingly. If using an audio cue in training (possible feature), we set this threshold based on how close to the cue they landed (maybe <0.1s = good, >0.3s = late). In absence of external reference, we interpret any split that clearly occurs *after* the movement as late. As we gather data, we might find users consistently land ~0.1s before moving – then 0.1s becomes our expected and >0.2 late. For now, 0.2s is a generic sweet spot.

- **Stance Width:** We measure ratio = feet distance / shoulder distance (both in same units, e.g., pixel length at some reference depth). We want roughly 1.0 (shoulder width) for a ready stance, and even a bit more (1.2–1.5) on lunges to maximize reach and stability. *Too narrow:* <0.8 in ready or landing. *Too wide:* >1.5 in ready (unlikely, but if someone stands super wide it might impede movement). For lunges, if ratio <0.7, definitely too narrow (almost feet together). Many amateurs lunge with feet nearly inline (one behind the other), so 0.7–0.8 catches that. We start with **Threshold: 0.8× shoulder width** as the line below which we flag "stance too narrow" on a landing or split. And perhaps >1.5× as "too wide" if ever seen (less common error but could happen if overextending). Calibration: This should be personalized. If a user is very tall or short, shoulder width vs foot stance might vary. We can calibrate by taking their average stance over the session: if someone naturally stands at 0.6 (maybe camera perspective made it appear smaller?), we adjust the threshold down a bit so we're not nagging their normal. Conversely if someone naturally uses 1.0 always, we might encourage going a bit wider for aggressive moves – but carefully. We can also do an initial calibration routine: ask user to stand in athletic stance for a moment, measure that ratio as their "comfortable stance = baseline", then set "too narrow" as, say, 85% of that baseline.

- **Knee Collapse Indicator:** This is tricky to quantify in 2D. We'll use a proxy: the angle between hip-knee-ankle as seen from camera (if camera is head-on or behind). Normally in a lunge, that should be roughly in line (~180° if no valgus). If the knee caves in, the angle at the knee (in frontal projection) becomes acute (<180). If it drops below ~160°, that's a red flag for valgus. So **Threshold: knee_angle_frontal < 160°** = potential knee collapse. Another measure: horizontal distance between knee and foot. If the knee's x is more than, say, 10% of hip width inside the foot's x position, that indicates inward movement. Since we likely can't get frontal view for that precisely, we rely on the angle. We will use **160°** as initial. Calibration: If camera angle is not frontal, this angle will appear smaller even if form is fine (side view yields a different projection). We thus only apply this if camera is roughly frontal or behind. If user videos are mostly from behind (which is likely recommended for footwork), the knee angle can be gauged. We might calibrate per user by comparing left vs right consistency, etc., but given difficulty, this will be a **soft rule**: if triggered, we mark "possible knee collapse – verify from front view."

- **Recovery-to-Base Timing:** For drills, an acceptable recovery (from lunge back to ready) might be ~1 second or less. In pro play, players often recover in under 1s to prepare for next shot. For training, if it's consistently >1.5s, that's slow (maybe fitness issue or not pushing off). So **Threshold: 1.0s** as target, >1.5s = "slow recovery" warning. We will mark each recovery duration, and any above 1.5s gets flagged. If all are ~1.3s, we might still encourage improving if they aim for competition standard. Calibration: If the user is older or beginner, 1.5 might be fine; we can adjust based on context (maybe ask user skill level in profile). We can also adjust after initial sessions – if user always 2s, maybe set goal to under 1.5 gradually. So eventually personalized goals. But 1.0s is a baseline threshold from general coaching benchmarks.

- **Balance/Landing Stability:** Not a single number threshold, but things to watch: extra steps or foot shuffles after landing, or large torso sway. A simple metric: how far the center of mass (like midpoint of hips) moves in the 0.5s after landing. If > e.g. 20% of person's height, that suggests a stumble or step. Also, if we detect any additional distinct foot contact events (like a small hop or adjustment step) within 1s after lunge, that means they weren't stable on landing. **Threshold: COM variance > 0.2 (normalized) or secondary foot movement** within 1s = unstable landing. Another indicator: if the user's foot that landed slides (which we might catch as the foot moving while low to ground). Hard to measure precisely with our data, but we set criteria and see. Calibration: This might be binary mostly. But for each lunge, if flagged unstable, we check if multiple lunges flagged – if yes, mention "balance could improve". If just one slip, maybe ignore. We can calibrate by requiring it in >20% of moves to call it a general issue.

- **Split Hop Height:** While not explicitly asked, we might monitor if the split step is too shallow (some players barely hop which reduces its effectiveness). If we had calibration like measuring how much knees bend or how much ankles lift: e.g., if ankles only moved up <5cm, that's a shallow split. For now, not a priority threshold, but it's one to keep in mind (could add later: "jump higher on split for more explosiveness").

Calibration Approach: For a new user, we could include a quick **calibration drill**: for example, ask them to do one or two lunges and recover at comfortable speed, and measure those values. Use those as baseline to define their thresholds at first, rather than generic values. If not, we start with above defaults and then after first full session, adjust the thresholds slightly towards the user's average minus some margin to gently push improvement without constantly flagging. E.g., if user's average stance width was 0.85 and we flagged <0.8, we saw 3 flags at 0.75. We might keep threshold 0.8 (since we do want them to reach at least 0.8). If user's average recovery 1.3s and we flagged >1.5, maybe only one flagged at 1.6s, that's fine. So initial thresholds likely remain until user consistently exceeds them or

never triggers them (if never triggered, maybe user is advanced – we could tighten to challenge them, but careful not to push into injury territory).

We will communicate these thresholds transparently whenever possible, so users know what the coach is expecting ("Try to recover under **1.5s**; you averaged 1.8s this time [5] "). These numbers can be refined with feedback from actual use (maybe 1.2s is more realistic for average amateur and our 1.0s was too strict, etc.).

5. Chat Grounding Plan

The Coach Chat should act like an experienced badminton coach who has watched the session, **but only knows what's in the data**. The plan to ensure it never hallucinates or gives generic platitudes:

Data Preparation for QA: We convert the analysis JSON into a set of retrievable documents (text passages) that the LLM can refer to. For example, we might create a summary for each event: "At 4.7s: Lunge (forehand front). Stance width 0.75× shoulder (too narrow). Knee angle 85° (good deep lunge). Issues: stance_narrow, torso_lean." and for metrics: "Overall stance width: min 0.75×, avg 0.88×. Narrow stances 3 times (e.g. 4.7s)." These become text snippets, possibly with keys or embedding vectors. We can store them in a simple in-memory list or use a vector store (like FAISS or Chroma) with embeddings. Given the scale (few dozen lines), even a simple keyword search would suffice.

Retrieval-Augmented Generation (RAG): When the user asks a question ("How was my stance?"), the system will: 1. **Parse the question** – possibly using an LLM to identify keywords ("stance"). But a simpler approach: we have predefined categories (stance, split timing, balance, etc.). We match "stance" to retrieving any passage about stance metrics or stance issues. 2. **Retrieve relevant data** – e.g., the snippet that says how often stance was narrow and the values. Also maybe retrieve events where stance was flagged. 3. **Compose prompt for LLM** – The prompt will include a system message like: "You are an AI badminton coach. You have data from the player's session. Only use that data to answer. Do not speculate beyond it." Then include the retrieved snippets as context. Possibly also include some coaching knowledge if needed (like we might have a knowledge snippet: "Ideal stance is around shoulder width for good balance." but ideally that could be pre-fed or the LLM might know that). But to be safe, we provide any such domain knowledge in the prompt as well rather than expecting the model to know (to avoid it making things up). 4. **LLM generates answer** – It will use the facts to create a friendly answer, citing specific times. For example: "Your stance tended to be narrow on some lunges. For instance, at **4.7s** your feet were only about 75% of shoulder width apart [5] . That's quite narrow – try to keep about shoulder-width to maintain balance. Overall, your average stance was ~0.88×, which is a bit on the low side. Aim for closer to 1.0× or slightly more." The citation [5] would link to the underlying data (in our UI, maybe clicking that highlights event at 4.7s). 5. **Guardrails and Verification:** We will implement a check on the LLM output: does it mention any fact/time that wasn't in the retrieved data? Tools like Guardrails can enforce that no unknown claims slip in. For instance, if the model said "You improved a lot compared to last time," but we gave it no data of last time, we'd catch that as hallucination. To be safe, we instruct it not to mention any session beyond current.

No Generic AI Advice: The coach should not spout general tips like "practice footwork drills" out of nowhere. Every suggestion must tie to a measured observation. If a user asks something broad like "How can I be faster on court?", the chat will still lean on data ("In this session, your recovery times averaged 1.4s. Improving that will make you faster in rallies [5] . One way to do so is to focus on pushing off immediately after lunging…"). We may allow the model to add known training tips (like "do ladder drills") but only if it logically follows from an observed weakness and is phrased as a suggestion, not a hallucinated fact.

Citing Timestamps: We enforce that whenever the AI references a specific moment or metric, it cites it. In a chat UI, these could link to the video or highlight on a chart. The citation format will use our internal reference system (which might be like 【event 2】 or a time stamp link). For the purpose of this answer, we showed it like the other references. For the actual product, the citation might just be "(4.7s)" clickable. The key is the user trusts that any claim "at X time something happened" is verifiable.

Handling Uncertainty: The chat will be trained/prompted to be honest about data uncertainty. For example, "It looks like your left knee might have buckled inward on a lunge at 4.7s, but the camera angle makes it a bit hard to be sure [5]. If possible, try a front view next time to check this." This way, it doesn't hallucinate confidence where we flagged something as low confidence.

LLM Selection: GPT-4 is a good choice because of its reliability in following instructions and writing coherent, encouraging prose. For cost saving, GPT-3.5 could suffice for simpler Q&A. Eventually, an open model fine-tuned on our domain (with conversations based on our data) could be deployed locally for privacy.

Testing the Chat: We will thoroughly test with some sample questions: - Direct metric Qs: "What was my average recovery time?" -> Should retrieve metrics.average_recovery_time and answer "1.1 seconds on average [5], which is a bit slow relative to target 1.0s." - Explanation Qs: "Why is a wide stance important?" -> The model should explain using context that we hopefully supply (maybe we include a snippet in knowledge: "Wider stance gives stability; narrow stance can cause imbalance."). If not, the model likely knows general sports knowledge. But to be safe, we might allow general knowledge here. It should still anchor to the user: "In your case, a narrow stance caused balance issues at 4.7s [5], which is why a wider stance would help." - Reassurance Qs: "Did I improve from last time?" -> If we have no data from last time, the assistant should respond carefully: "I only have data from this session. I can't compare to previous sessions right now [5]." (Unless we store history; that's a future feature.) - Off-topic Qs: If user asks something completely unrelated (or tries to get it to coach them on another sport), we restrict it to say it can't answer beyond session context.

Never Beyond JSON: We will explicitly instruct the LLM that if the user asks about something not in the session data (like "How do I smash harder?" which is stroke technique not measured by footwork), the AI coach should either politely say that's outside this analysis, or give a very generic answer but labeled as general tip, not personal. Possibly: "I don't have data on your strokes, but generally smashing harder comes from technique and timing. However, based on your footwork, improving your positioning might indirectly help your smash." Always transparent that it's not from session evidence.

Privacy in Chat: Since we're not storing raw video or identifiable data, the chat won't have sensitive info aside from performance. Still, we ensure that any personal data (name, etc., which might be in session_info) is handled appropriately. But that's minor here.

In summary, the chat's brain is the JSON analysis, and we will **algorithmically leash** the AI to that data. It becomes more of a natural language renderer of our stats, with a bit of coaching wisdom in tone. This approach ensures trustworthiness – the user can see "the AI isn't making this up, it has receipts for every critique it gives me."

6. 1–2 Week Build Plan (Execution Checklist)

We outline the first 1-2 weeks of execution, focusing on a functional prototype that validates the core features. Each task is defined with an acceptance test:

Week 1: Core Pipeline & Basic Output

1. **Set up Development Environment** – Set up repository, Cloud Run project, get MediaPipe and TF working in Python. *Acceptance:* Can run a test script that loads a pose model and processes one image (verifying dependencies).
2. **Pose Extraction Module** – Write code to read a video (use a short sample of badminton movement), run the pose model on each frame (or every Nth frame for speed initially), and output raw keypoints with timestamps. *Acceptance:* Running on a 10s sample video yields a list of keypoint arrays for each frame, and it does so in reasonable time (e.g., 10s video in <15s). Check that keypoints look plausible (e.g., by printing first few or plotting one frame skeleton).
3. **Apply Smoothing Filter** – Implement OneEuro and apply to the sequence of keypoints. *Acceptance:* Verify on a jittery sequence (maybe add noise to keypoints) that the filter smooths it. Specifically, log one joint coordinate pre- and post-filter to see reduction in jitter amplitude. Also confirm minimal lag on a quick movement (oneEuro's job).
4. **Event Detection Logic (Initial)** – Implement a simplified FSM for just one or two events, say detect lunges and recoveries. Use simple thresholds to start (distance and time). *Acceptance:* On a sample video where the user does a clear lunge and return, the code identifies one "Lunge" event and one "Recovery" event at roughly correct times. Print them out. No false events in between.
5. **Data Structuring** – Assemble the JSON output with those events and maybe a dummy metric or two. *Acceptance:* After processing sample video, we can output a JSON file that matches the schema (with placeholder values if needed). Validate JSON format.

(By end of Week 1, we have a back-end script that can take a video and produce a rudimentary JSON analysis. Possibly no front-end yet, but we can manually inspect JSON.)

Week 2: Front-End & Feedback Integration

6. **Simple Front-End Page** – Create a basic web page where user can upload a video and then see the output JSON or a simple text report. Use a minimal backend endpoint (Flask or FastAPI) to accept file and run analysis (from Week 1 code). *Acceptance:* Running locally or on a test server, we can upload a known video and after processing, the page displays "Lunge at 4.7s, Recovery at 6.0s…" (just textual for now).
7. **Visual Overlay Prototype** – Using the canvas or video element, draw something for the events. For now, maybe highlight frames at event times. *Acceptance:* On the web page, there is a video player that can play the uploaded video (or we use a fixed test video in the page for now), and markers on a timeline or a list of event timestamps that when clicked, jump the video. This proves we can link analysis to video.
8. **Integrate Chat (Basic)** – Hook up a simple retrieval QA. Could be as simple as a hardcoded QA: user types "How many lunges?" and we parse JSON to answer "5 lunges." But ideally, use LangChain with a local LLM or OpenAI API. *Acceptance:* If the user types a question about the data, the system returns a relevant answer referencing the JSON. Test with a few known queries ("When was my narrow stance?" -> it should find event with stance_narrow and respond). This doesn't have to be fully polished, just demonstrate the concept.
9. **One-Cue Feedback Mechanism** – Implement in analysis logic: ensure if multiple issues occur in one event, we mark one as primary. E.g., choose stance_narrow over torso_lean if both happen, or vice versa, to emphasize one. And overall, ensure our JSON's overall_issues is prioritized. *Acceptance:* Review JSON to confirm for each event only one main issue is highlighted, and overall_issues summary picks up distinct categories, not spamming everything.
10. **Test on Realistic Footage** – Take a sample smartphone video of a person doing a badminton footwork drill (if available – could simulate with another sport or a mock-up if not). Run the system end-to-end. *Acceptance:* The events detected align reasonably with actual actions in the video (no glaring misses or false alarms). The output JSON and any UI elements reflect correct info. For example, if

person did 3 lunges, the system shouldn't output 0 or 10. Also check performance: processing doesn't take ridiculously long or crash memory.

Acceptance Tests & "Definition of Done" for Prototype: - User can upload a ~1 minute video and within ~1-2 minutes get back a structured report (maybe not beautifully formatted yet, but content is there). - The report identifies at least one type of event (lunge) and one metric (count of lunges or average stance). - If the user asks via chat interface "What can I improve?", the system returns an answer referencing one of the issues found (even if simple). - All data stays local or on our server (no video gets sent to unauthorized third parties). - No major crashes or obvious mis-labeling of events in tested scenarios.

Stretch goals for those 2 weeks (if ahead): - Add voice feedback in real-time for a live camera input (just on a dev machine) to test latency. - Deploy the prototype on actual Cloud Run and test with a mobile-recorded video. - Collect feedback from a badminton coach on the usefulness of the feedback phrasing.

By focusing on these tasks, in 1-2 weeks we aim to have a **minimum viable "ShuttleSense"**: you upload a video and get a basic coaching report. From there, we can iteratively refine accuracy (tweak thresholds, add event types like split detection), improve the UI (graphs, nicer chat), and expand capabilities (multi-session tracking, etc.). The key is that within 2 weeks, we should demonstrate end-to-end functionality – that will guide where to focus next (e.g., if pose accuracy is a problem, or if the chat is underwhelming, etc., we'll address accordingly).

Recommended Next Step: Set up a test with real user footage to validate the detection accuracy and refine the thresholds before expanding to more features. 43 5

---

1 MoveNet: Ultra fast and accurate pose detection model. | TensorFlow Hub
https://www.tensorflow.org/hub/tutorials/movenet

2 18 19 20 40 Setting-up Smoothing Filters for MediaPipe Pose Estimation Pipeline using Python: A Practical Guide | by Debasish Raut | Nov, 2025 | Medium
https://medium.com/@debasishraut.dev/setting-up-smoothing-filters-for-mediapipe-pose-estimation-pipeline-a-practical-guide-fcc03f462196

3 9 10 11 13 41 42 43 Enhanced Pose Estimation for Badminton Players via Improved YOLOv8-Pose with Efficient Local Attention - PMC
https://pmc.ncbi.nlm.nih.gov/articles/PMC12298368/

4 The Nature of Coaching Cues | Mark Rippetoe - Starting Strength
https://startingstrength.com/article/the-nature-of-coaching-cues

5 Exploring Pose-Based Anomaly Detection for Retail Security: A Real-World Shoplifting Dataset and Benchmark
https://openaccess.thecvf.com/content/WACV2025W/ASTAD/papers/Rashvand_Exploring_Pose-Based_Anomaly_Detection_for_Retail_Security_A_Real-World_Shoplifting_WACVW_2025_paper.pdf

6 casiez/OneEuroFilter: Algorithm to filter noisy signals for ... - GitHub
https://github.com/casiez/OneEuroFilter

7 MMPose - OpenMMLab Pose Estimation Toolbox
https://mmpose.com/

8 24 Pose Detection Showdown: BlazePose, MoveNet & YOLOv11 | Kite Metric
https://kitemetric.com/blogs/open-source-pose-detection-a-deep-dive-into-blazepose-movenet-and-yolov11

12 Guide to OpenPose: Real-Time Multi-Person Detection - Viso Suite
https://viso.ai/deep-learning/openpose/

14 MVIG-SJTU/AlphaPose: Real-Time and Accurate Full-Body ... - GitHub
https://github.com/MVIG-SJTU/AlphaPose

15 Source code for mmhuman3d.core.filter.oneeuro_filter
https://mmhuman3dyl-1993.readthedocs.io/en/latest/_modules/mmhuman3d/core/filter/oneeuro_filter.html

16 17 27 GitHub - giaongo/RepDetect: AI Android application crafted with Kotlin that harnesses the power of MediaPipe Pose Landmark Detection to deliver real-time feedback on exercise form while accurately counting repetitions
https://github.com/giaongo/RepDetect

21 31 32 33 GitHub - deepaktalwardt/badminton-pose-analysis: Badminton Pose Analysis for coaching and pose correction
https://github.com/deepaktalwardt/badminton-pose-analysis

22 Rules-Based Real-Time Gait Event Detection Algorithm for Lower ...
https://www.mdpi.com/1424-8220/22/22/8888

23 (PDF) A Deep Learning Approach to Badminton Player Footwork Detection Based on YOLO Models: A Comparative Study
https://www.researchgate.net/publication/384381848_A_Deep_Learning_Approach_to_Badminton_Player_Footwork_Detection_Based_on_YOLO_Models_A_Comparative_Study

25 tfjs-models/pose-detection/src/movenet/README.md at master
https://github.com/tensorflow/tfjs-models/blob/master/pose-detection/src/movenet/README.md

26 GitHub - yakupzengin/fitness-trainer-pose-estimation: An AI-powered fitness tracker that uses real-time pose estimation to count reps, monitor form, and provide instant feedback for exercises like squats, push-ups, and bicep curls. Designed for accuracy, motivation, and adaptability
https://github.com/yakupzengin/fitness-trainer-pose-estimation

28 OpenCap: Human movement dynamics from smartphone videos
https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1011462

29 [PDF] OpenCap Q&A - Mobilize Center
https://mobilize.stanford.edu/wp-content/uploads/2022/12/OpenCap-QA-Final.pdf

30 [PDF] OpenCap: 3D human movement dynamics from smartphone videos
https://nmbl.stanford.edu/wp-content/uploads/OpenCapManuscript_higherRes.pdf

34 Doubles Badminton Analysis with Singles-Trained Models - arXiv
https://arxiv.org/html/2508.13507v1

35 The analysis of motion recognition model for badminton player ...
https://www.nature.com/articles/s41598-025-02771-9

36 An integrated deep learning framework for real-time badminton ...
https://www.tandfonline.com/doi/full/10.1080/1448837X.2025.2553948?src=

37 [bug] pip version not updated · Issue #566 · guardrails-ai ... - GitHub
https://github.com/guardrails-ai/guardrails/issues/566

38 Features - Clutch - The AI Camera for Padel
https://www.clutchapp.io/features

39 RTMPose: Real-Time Multi-Person Pose Estimation based on ...
https://arxiv.org/abs/2303.07399