AI-Powered Virtual Tailor — Design & Architecture Document

Prepared for: Project Stakeholders
Date: November 27, 2025

Executive Summary

This document describes the system architecture, components, data flows, APIs, algorithms, and deployment plan for an AI-Powered Virtual Tailor web application. The solution integrates a React/Vite frontend with live camera capture and voice guidance, and a Python backend (FastAPI recommended) that processes auto-captured images to produce accurate body measurements using YOLOv8 segmentation and MediaPipe pose landmarks.

Goals & Scope

- Provide a modern web app that captures a user's front and side photos automatically using live webcam input. - Use voice guidance (TTS) to coach the user into a 1-meter distance and correct poses, and give capture feedback. - Backend processing must produce measurements in cm using pixel-to-unit conversion (PPU) and return images overlaid with masks, landmarks, and measurement lines. - Deliver a REST API that returns JSON with measurements, PPU, and Base64-encoded annotated images.

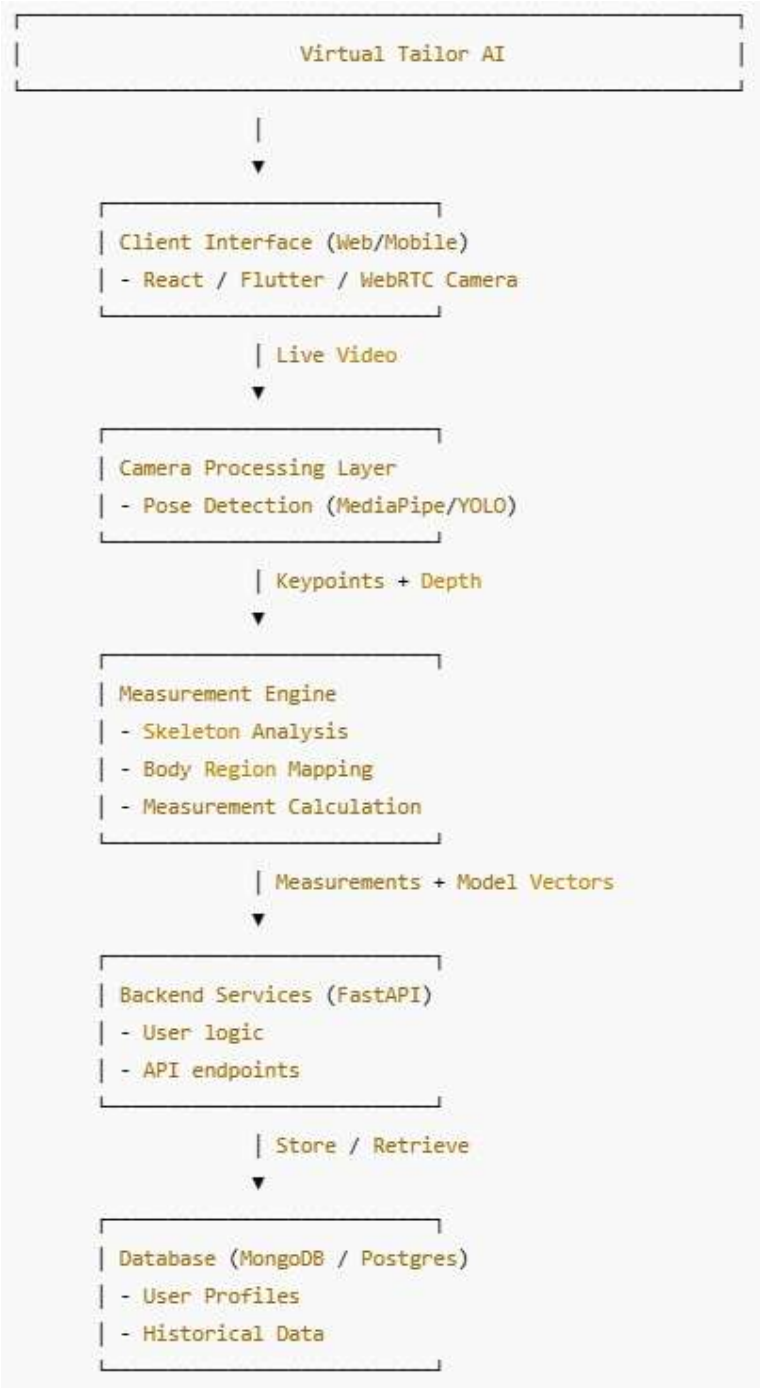
Assumptions & Constraints

1. Users will run the app in a modern browser supporting WebRTC and the Speech Synthesis API. 2. The user will follow guidance and stand approximately 1 meter from the camera; however, the system supports optional manual height input for improved PPU. 3. YOLOv8 segmentation and MediaPipe pose run in the backend (or partly on-device) depending on performance. 4. The environment has adequate lighting and a plain background for better segmentation. 5. Privacy: images are uploaded to the backend only for processing; consider on-device inference if privacy is critical.

High-Level Architecture

The system follows a client-server architecture with optional on-device preprocessing. Main components: - Frontend (React + Vite): Camera UI, TTS guidance, auto-capture logic, preview, and result visualization. - Backend (FastAPI): REST endpoints to accept images, run YOLOv8 segmentation, run MediaPipe pose, compute PPU and measurements, and return annotated images and JSON results. - Model storage & inference service: Containerized models (YOLOv8, MediaPipe), GPU-enabled inference nodes (optional). - Storage: Temporary object store for uploaded images and results (S3-compatible) and optionally a database for audit/logs. Monitoring & CI/CD: Docker, Kubernetes/ECS, Prometheus + Grafana, and CI pipelines.

Architecture Diagram (textual)



Frontend Design

Key responsibilities: 1. Live Camera Module: Use getUserMedia + WebRTC to show live preview. Implement a lightweight preview overlay to show skeleton/pose feedback (optional local MediaPipe) for immediate user feedback.

2. TTS & Voice Guidance: Use the browser Speech Synthesis API for voice prompts. Provide a queue of messages and interruptible prompts. Example prompts are provided in the requirements.

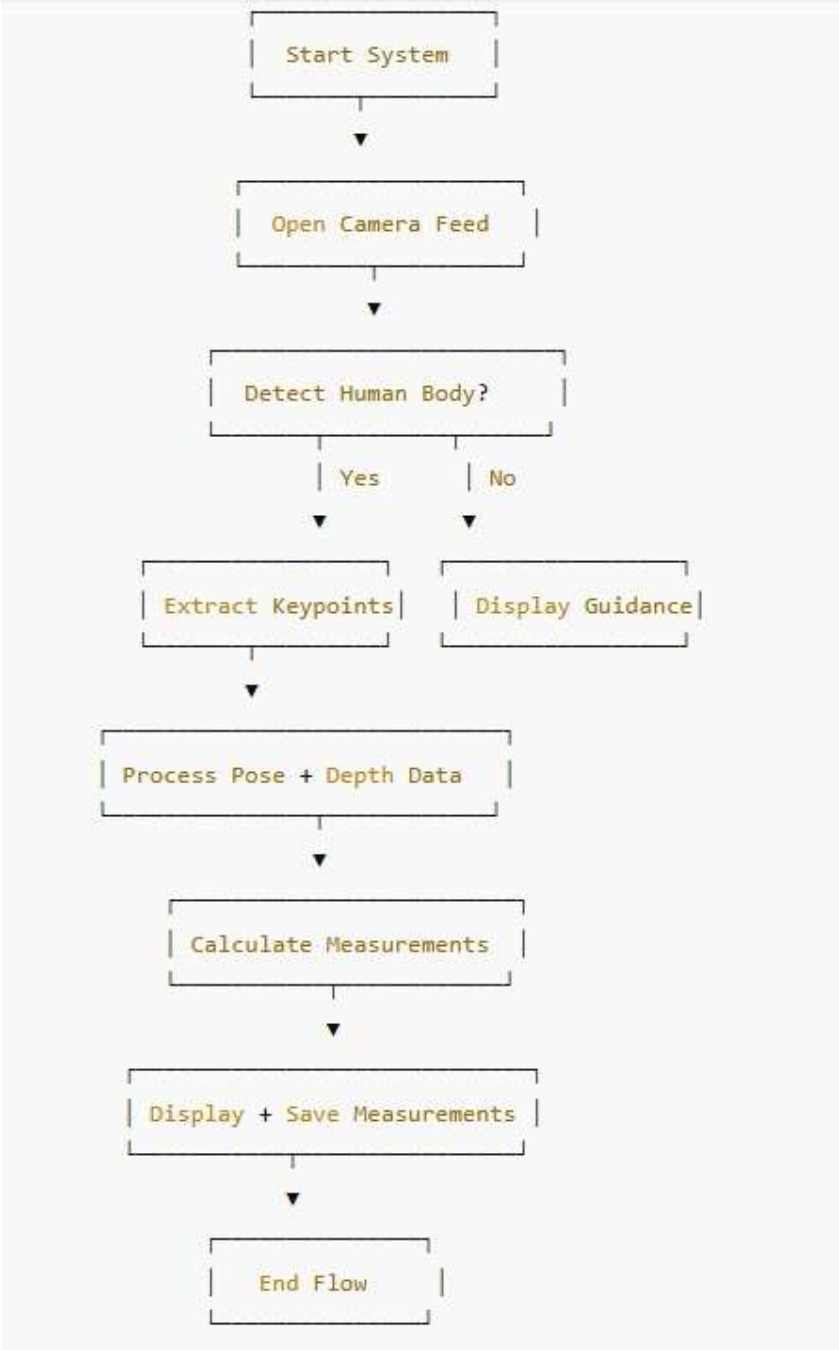
3. Auto-Capture Logic: Continuously process frames (e.g., every 100 ms) to obtain pose landmarks either locally (MediaPipe JS) or by sending low-res frames to a local inference worker. - Distance check: Estimate distance using shoulder pixel width relative to frame width or use a known reference object size if present. - Pose check:
Confirm A-pose/front or side profile via landmark orientation and visibility. - Stability check: Compute running variance of critical landmark positions over a short buffer (e.g., 2s); if below threshold, trigger capture.

4. Capture Flow: Sequential captures for front and side. Show progress, voice prompts, and visual countdown. 5. Results UI: Show original + annotated images side-by-side, an editable measurements table, and export options (PDF/CSV).

Backend Design

Recommended framework: FastAPI (async), Uvicorn/Gunicorn worker model. Components: - Capture endpoint: POST /api/v1/measurements - Health endpoint: GET /api/v1/health Status/webhook or job queue for long-running inferences (optional) - Inference worker(s): Containerized services that load YOLOv8 segmentation and use MediaPipe (python) for pose estimation. - Storage: Temporary S3 (or local) storage for uploaded images; Redis for caching and queue management.

Flow Chart



API Endpoints (brief)

Measurement Pipeline & Algorithms

1. Segmentation: Run YOLOv8 segmentation to obtain a human mask. Use the mask to remove background and for better circumference approximations.

2. Landmark Detection: Use MediaPipe Pose to extract 2D landmarks (shoulders, hips, waistline estimate point, knees, ankles, wrists).

3. Pixel-to-Unit (PPU) Calculation: - Option A (user height provided): PPU = height_pixels / height_cm. - Option B (1m guidance + camera FOV): Use distance estimates from shoulder width in pixels compared to expected shoulder width for 1m distance or a camera intrinsic FOV approximation. (Recommended: allow user-entered height for accuracy.)

4. Measurement computation: - Linear distances: measure pixel distance between two landmarks and convert using PPU. - Circumferences: approximate from width measurements and segmentation mask (e.g., hip width to circumference via elliptical/cylindrical assumptions or multiplying by empirically tuned factors). - Height: use landmark positions (top-of-head to heel) within mask to compute height_pixels.

5. Stability and pose classification: compute landmark visibility and angles (e.g., torso angle) to confirm front vs side.

Security, Privacy & Compliance

- Use HTTPS for all communications and strict CORS policy. - Minimize storage of raw images; delete after processing or store encrypted if retention is required. - Offer an on-device inference option to keep images local (MediaPipe + client-side YOLO if feasible). - Authentication: OAuth2 / JWT for registered users; ephemeral tokens for anonymous captures. - Log only metadata; avoid storing PII with images unless user consents.

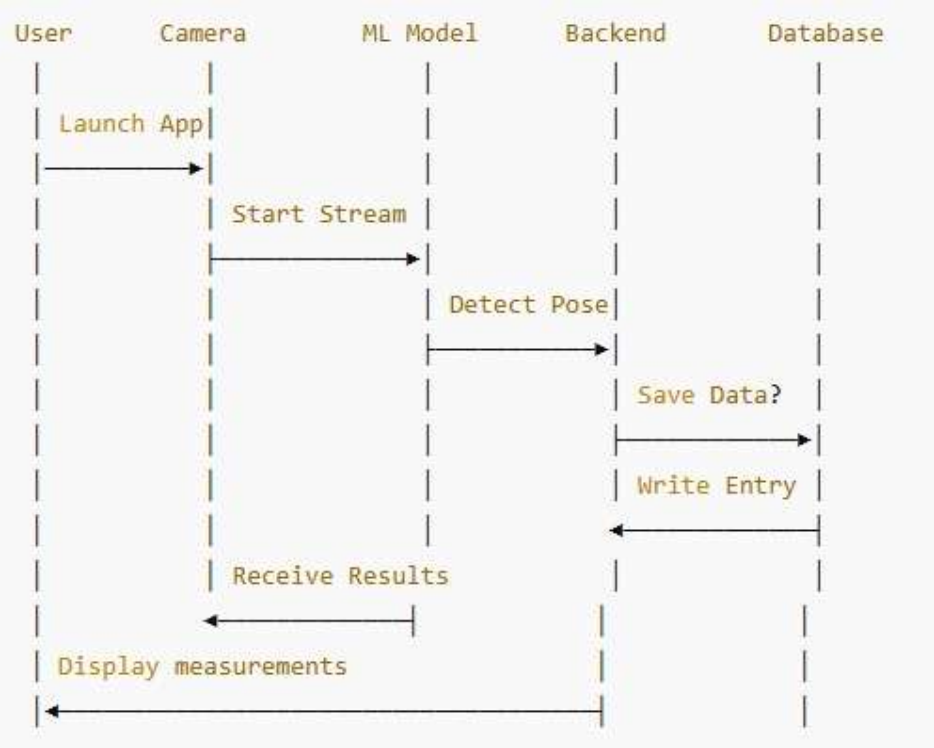
Scalability & Deployment

Deployment options: - Containerize backend and inference workers with Docker; use GPUs where available (NVIDIA GPU + CUDA for YOLOv8). - Orchestrate with Kubernetes (K8s) and use an autoscaling inference pool for burst traffic. - Use S3-compatible storage (or cloud buckets), Redis for queues, and a relational DB for user records. - CI/CD: GitHub Actions / GitLab CI to build images and run tests; run integration tests including sample captures and end-to-end measurement validation.

Testing, Validation & Metrics

- Unit tests for measurement math, PPU conversion, and API contract tests. - Integration tests: simulate camera captures and ensure the pipeline returns consistent measurements. - Accuracy validation: collect labeled ground-truth measurements from test subjects and compute error statistics (MAE, RMSE). Aim for <2.5 cm MAE on chest/waist in controlled conditions. Performance metrics: target 1–3 seconds processing time for inference on GPU; fallback CPU times will be longer.

Sequence Flow (Capture -> Result)



Implementation Checklist (MVP)

Frontend: - Camera preview + permission flow - Speech Synthesis guidance queue - Local pose check + stability buffer - Auto-capture and POST to backend Backend: - FastAPI app + endpoints - YOLOv8 segmentation inference - MediaPipe Pose inference - Measurement math and JSON response DevOps: - Dockerfile(s) for backend and inference - CI pipeline and simple K8s manifests