FitTwin MediaPipe MVP - Technical Specification (Speckit)

1. Strategic Vision and Architecture

1.1. DMaaS Platform Vision

FitTwin is being built as a **Data-Model-as-a-Service (DMaaS)** platform, not a consumer-facing app. The primary customers are AI systems (like ChatGPT-powered shopping assistants) and online retailers who need accurate body measurements and size recommendations via API.

Target Customers: - **Primary:** AI systems and online retailers (B2B API customers) - **Secondary:** General online shoppers (B2C end users)

Value Proposition: - For AI/Retailers: Robust API returning structured JSON of body measurements and size recommendations (high-value, high-margin) - For Consumers: Free, accurate body measurements without expensive 3D scanning hardware

1.2. MediaPipe-Only MVP Strategy

The MVP will launch with **MediaPipe-only measurement extraction**, eliminating per-scan vendor costs and maximizing data ownership from day one.

Key Strategic Decisions: - Use MediaPipe Pose Landmarker v3.1 for landmark detection - Calculate anthropometric measurements using geometric equations - Store all raw data (photos, landmarks, measurements) for proprietary model training - Add vendor API (3DLOOK or Nettelo) as **optional fallback** only if accuracy <97% (>3% error) - Target accuracy: <3% measurement error for core dimensions (height, chest, waist, hip, inseam)

Benefits: - Zero per-scan costs - Full data ownership and IP control - Rapid iteration without vendor dependencies - Clear path to proprietary model training

1.3. Hybrid Measurement Strategy (Simplified)

The original three-layer hybrid strategy has been simplified to two stages for the MVP:

Stage	Technology	Purpose	Cost	Status
Stage 1	MediaPipe Pose Landmarker v3.1	Free, immediate measurement extraction	$0.00 per user	**MVP Launch**
Stage 2	Optional Vendor API (3DLOOK/Nettelo)	Calibration if accuracy <97%	50–200 total for 50–100 scans	**Conditional**

If MediaPipe accuracy proves sufficient (⩾97%), Stage 2 is skipped entirely. If not, 50–100 calibration scans are purchased to fine-tune the geometric equations.

---

2. Technical Architecture Overview

2.1. System Components

The FitTwin system consists of four main components:

1. **Frontend (Flutter/SwiftUI):** iPhone app for two-photo capture (front and side)
2. **MediaPipe Integration:** On-device landmark detection and measurement calculation
3. **Backend (FastAPI):** API for measurement validation, recommendation, and data storage
4. **Database (Supabase):** Postgres database with RLS policies for secure data storage
5. **Agent System (CrewAI):** Multi-agent system for autonomous development and orchestration

2.2. Data Flow

The end-to-end data flow for the MVP is:

1. **User captures two photos** (front and side) using iPhone camera
2. **MediaPipe detects landmarks** on-device using Pose Landmarker v3.1
3. **Local calculation** of approximate anthropometric measurements using geometric equations
4. **App sends data to backend** (photos, landmarks, measurements) via `/measurements/validate` endpoint
5. **Backend validates and normalizes** measurements, stores in Supabase with full provenance
6. **Backend generates size recommendations** using fit rules and returns via `/measurements/recommend` endpoint
7. **App displays results** to user and optionally sends to DMaaS API for external consumption

Optional Calibration Flow (if accuracy <97%): - Backend flags low-confidence measurements - CEO Agent triggers vendor API call (3DLOOK or Nettelo) - Vendor measurements stored separately for calibration only - ML Engineer Agent uses calibration data to refine geometric equations - Updated equations deployed to production

2.3. Technology Stack

Component	Technology	Rationale
Frontend	Flutter/SwiftUI	Cross-platform (Flutter) or native iOS (SwiftUI) for best performance
Measurement	MediaPipe Pose Landmarker v3.1	Free, accurate, on-device, no vendor lock-in
Backend	FastAPI (Python)	Fast, modern, async, excellent for ML integration
Database	Supabase (Postgres)	Free tier, RLS policies, real-time, excellent DX
Agents	CrewAI	Multi-agent orchestration, LLM-powered reasoning
CI/CD	GitHub Actions	Free for public repos, excellent integration
Deployment	TestFlight (iOS), Supabase Edge Functions (API)	Low-cost, production-ready

---

3. Backend Implementation

3.1. Error Response Schema

Create a consistent error envelope that agents and external API consumers can parse reliably.

File: `backend/app/schemas/errors.py` (NEW)

```
from pydantic import BaseModel
from typing import List, Optional

class ErrorDetail(BaseModel):
    """Detailed error information for a specific field or issue."""
    field: str
    message: str
    hint: Optional[str] = None

class ErrorResponse(BaseModel):
    """Consistent error response envelope for all API errors."""
    type: str  # e.g., "validation_error", "server_error",
"authentication_error"
    code: str  # e.g., "schema", "unit", "unknown_field", "accuracy_threshold"
    message: str
    errors: List[ErrorDetail] = []
    session_id: Optional[str] = None
```

3.2. Measurement Schema

Define the canonical measurement names, unit enum, and schemas for input, MediaPipe landmarks, and normalized output.

File: `backend/app/schemas/measure_schema.py` (UPDATE or CREATE)

```python
from pydantic import BaseModel, Field, validator
from enum import Enum
from typing import Optional, List, Dict

class Unit(str, Enum):
    """Supported measurement units."""
    CM = "cm"
    IN = "in"

class MediaPipeLandmark(BaseModel):
    """Single MediaPipe landmark with 3D coordinates."""
    x: float
    y: float
    z: float
    visibility: float

class MediaPipeLandmarks(BaseModel):
    """Complete set of MediaPipe Pose landmarks."""
    landmarks: List[MediaPipeLandmark]
    timestamp: str
    image_width: int
    image_height: int

class MeasurementInput(BaseModel):
    """Input schema for measurements with flexible units and MediaPipe data."""
    # Core measurements (optional, can be calculated from landmarks)
    height: Optional[float] = None
    neck: Optional[float] = None
    shoulder: Optional[float] = None
    chest: Optional[float] = None
    underbust: Optional[float] = None
    waist_natural: Optional[float] = None
    sleeve: Optional[float] = None
    bicep: Optional[float] = None
    forearm: Optional[float] = None
    hip_low: Optional[float] = None
    thigh: Optional[float] = None
    knee: Optional[float] = None
    calf: Optional[float] = None
    ankle: Optional[float] = None
    front_rise: Optional[float] = None
    back_rise: Optional[float] = None
    inseam: Optional[float] = None
    outseam: Optional[float] = None

    # Unit and metadata
    unit: Unit = Unit.CM
    session_id: Optional[str] = None

    # MediaPipe data
    front_landmarks: Optional[MediaPipeLandmarks] = None
    side_landmarks: Optional[MediaPipeLandmarks] = None

    # Photo URLs (for storage and future model training)
    front_photo_url: Optional[str] = None
    side_photo_url: Optional[str] = None

    @validator(\'*\', pre=True)
    def check_positive(cls, v, field):
        """Validate that numeric measurements are positive."""
        if field.name in [\'unit\', \'session_id\', \'front_landmarks\',
\'side_landmarks\',
                          \'front_photo_url\', \'side_photo_url\']:
            return v
        if v is not None and isinstance(v, (int, float)) and v <= 0:
            raise ValueError(f"{field.name} must be positive")
        return v

class MeasurementNormalized(BaseModel):
    """Normalized measurement schema (all in cm) with confidence scores."""
    height_cm: float
    neck_cm: float
    shoulder_cm: float
    chest_cm: float
    underbust_cm: float
    waist_natural_cm: float
    sleeve_cm: float
    bicep_cm: float
    forearm_cm: float
    hip_low_cm: float
    thigh_cm: float
    knee_cm: float
    calf_cm: float
    ankle_cm: float
    front_rise_cm: float
    back_rise_cm: float
    inseam_cm: float
    outseam_cm: float

    # Metadata
    source: str = "mediapipe"  # "mediapipe", "user_input", "vendor_api"
    model_version: str = "v1.0-mediapipe"
    confidence: float = Field(ge=0, le=1, default=1.0)
    accuracy_estimate: Optional[float] = None  # Estimated % error
    session_id: Optional[str] = None

    # Provenance
    front_photo_url: Optional[str] = None
    side_photo_url: Optional[str] = None
    front_landmarks_id: Optional[str] = None  # Reference to stored landmarks
    side_landmarks_id: Optional[str] = None
```

3.3. Validation and Normalization Utility

Implement the normalization and validation logic, including MediaPipe landmark processing.

File: `backend/app/core/validation.py` (NEW)

```python
 from backend.app.schemas.measure_schema import (
    MeasurementInput, MeasurementNormalized, MediaPipeLandmarks, Unit
)
from backend.app.schemas.errors import ErrorResponse, ErrorDetail
from fastapi import HTTPException
from typing import Dict
import uuid

CANONICAL_FIELDS = {
    "height", "neck", "shoulder", "chest", "underbust", "waist_natural",
    "sleeve", "bicep", "forearm", "hip_low", "thigh", "knee",
    "calf", "ankle", "front_rise", "back_rise", "inseam", "outseam"
}

def inches_to_cm(inches: float) -> float:
    """Convert inches to centimeters."""
    return inches * 2.54

def calculate_measurements_from_landmarks(
    front_landmarks: MediaPipeLandmarks,
    side_landmarks: MediaPipeLandmarks
) -> Dict[str, float]:
    """
    Calculate anthropometric measurements from MediaPipe landmarks.

    This function implements geometric equations to estimate body measurements
    from 3D landmark coordinates. The equations are based on anthropometric
    research and will be refined during the calibration phase if needed.

    Args:
        front_landmarks: MediaPipe landmarks from front-facing photo
        side_landmarks: MediaPipe landmarks from side-facing photo

    Returns:
        Dictionary of measurement names to values in centimeters
    """
    # TODO: Implement geometric equations
    # For MVP, this will include:
    # - Height: Distance from ankle to top of head
    # - Shoulder width: Distance between shoulder landmarks
    # - Chest: Circumference estimate from shoulder-to-hip distance
    # - Waist: Circumference estimate from hip landmarks
    # - Inseam: Distance from ankle to hip
    # etc.

    # Placeholder implementation
    measurements = {
        "height_cm": 170.0,
        "neck_cm": 38.0,
        "shoulder_cm": 45.0,
        "chest_cm": 100.0,
        "underbust_cm": 85.0,
        "waist_natural_cm": 80.0,
        "sleeve_cm": 60.0,
        "bicep_cm": 30.0,
        "forearm_cm": 25.0,
        "hip_low_cm": 100.0,
        "thigh_cm": 55.0,
        "knee_cm": 38.0,
        "calf_cm": 35.0,
        "ankle_cm": 22.0,
        "front_rise_cm": 25.0,
        "back_rise_cm": 35.0,
        "inseam_cm": 76.0,
        "outseam_cm": 100.0,
    }

    return measurements

def estimate_accuracy(measurements: Dict[str, float]) -> float:
    """
    Estimate the accuracy of MediaPipe-derived measurements.

    This function uses heuristics to estimate the % error of measurements
    based on landmark visibility, pose quality, and consistency checks.

    Returns:
        Estimated accuracy as a percentage (e.g., 0.97 for 97% accuracy, 3%
error)
    """
    # TODO: Implement accuracy estimation logic
    # For MVP, this will check:
    # - Landmark visibility scores
    # - Pose symmetry (left vs right side)
    # - Consistency between front and side measurements
    # - Known anthropometric ratios (e.g., height vs inseam)

    # Placeholder: assume 95% accuracy for MVP
    return 0.95

def normalize_and_validate(input_data: MeasurementInput) ->
MeasurementNormalized:
    """
    Normalize measurement input to centimeters and validate field names.

    If MediaPipe landmarks are provided, calculate measurements from landmarks.
    Otherwise, use user-provided measurements.

    Raises HTTPException with 422 status for validation errors.
    """
    errors = []

    # Check for unknown fields in user-provided measurements
    if not input_data.front_landmarks and not input_data.side_landmarks:
        input_dict = input_data.dict(exclude_unset=True)
        for field_name in input_dict.keys():
            if field_name not in CANONICAL_FIELDS and field_name not in [
                "unit", "session_id", "front_landmarks", "side_landmarks",
                "front_photo_url", "side_photo_url"
            ]:
                errors.append(ErrorDetail(
                    field=field_name,
                    message=f"Unknown field: {field_name}",
                    hint=f"Did you mean one of: {', '.join(CANONICAL_FIELDS)}?"
                ))

    if errors:
        raise HTTPException(
            status_code=422,
            detail=ErrorResponse(
                type="validation_error",
                code="unknown_field",
                message="One or more fields are not recognized",
                errors=errors,
                session_id=input_data.session_id
            ).dict()
        )

    # Calculate measurements from MediaPipe landmarks if available
    if input_data.front_landmarks and input_data.side_landmarks:
        measurements = calculate_measurements_from_landmarks(
            input_data.front_landmarks,
            input_data.side_landmarks
        )
        source = "mediapipe"
        accuracy = estimate_accuracy(measurements)

        # Store landmarks for provenance
        front_landmarks_id = str(uuid.uuid4())
        side_landmarks_id = str(uuid.uuid4())
        # TODO: Store landmarks in database

    else:
        # Use user-provided measurements
        conversion_factor = 2.54 if input_data.unit == Unit.IN else 1.0
        measurements = {
            "height_cm": (input_data.height or 0) * conversion_factor,
            "neck_cm": (input_data.neck or 0) * conversion_factor,
            "shoulder_cm": (input_data.shoulder or 0) * conversion_factor,
            "chest_cm": (input_data.chest or 0) * conversion_factor,
            "underbust_cm": (input_data.underbust or 0) * conversion_factor,
            "waist_natural_cm": (input_data.waist_natural or 0) *
conversion_factor,
            "sleeve_cm": (input_data.sleeve or 0) * conversion_factor,
            "bicep_cm": (input_data.bicep or 0) * conversion_factor,
            "forearm_cm": (input_data.forearm or 0) * conversion_factor,
            "hip_low_cm": (input_data.hip_low or 0) * conversion_factor,
            "thigh_cm": (input_data.thigh or 0) * conversion_factor,
            "knee_cm": (input_data.knee or 0) * conversion_factor,
            "calf_cm": (input_data.calf or 0) * conversion_factor,
            "ankle_cm": (input_data.ankle or 0) * conversion_factor,
            "front_rise_cm": (input_data.front_rise or 0) * conversion_factor,
            "back_rise_cm": (input_data.back_rise or 0) * conversion_factor,
            "inseam_cm": (input_data.inseam or 0) * conversion_factor,
            "outseam_cm": (input_data.outseam or 0) * conversion_factor,
        }
        source = "user_input"
        accuracy = 1.0  # Assume user input is accurate
        front_landmarks_id = None
        side_landmarks_id = None

    # Create normalized measurement object
    normalized = MeasurementNormalized(
        **measurements,
        source=source,
        model_version="v1.0-mediapipe",
        confidence=accuracy,
        accuracy_estimate=1.0 - accuracy,  # Convert to error percentage
        session_id=input_data.session_id,
        front_photo_url=input_data.front_photo_url,
        side_photo_url=input_data.side_photo_url,
        front_landmarks_id=front_landmarks_id,
        side_landmarks_id=side_landmarks_id,
    )

    return normalized
```

3.4. Measurement Router

Create the new measurement router with validation and recommendation endpoints.

File: `backend/app/routers/measurements.py` (NEW)

```python
from fastapi import APIRouter, HTTPException, Header
from backend.app.schemas.measure_schema import MeasurementInput,
MeasurementNormalized
from backend.app.schemas.models import RecommendationModel
from backend.app.schemas.errors import ErrorResponse, ErrorDetail
from backend.app.core.validation import normalize_and_validate
from backend.app.services.fit_rules_tops import recommend_top
from backend.app.services.fit_rules_bottoms import recommend_bottom
from typing import List, Optional
import os

router = APIRouter(prefix="/measurements", tags=["measurements"])

# Simple API key check (for staging)
VALID_API_KEY = os.getenv("API_KEY", "staging-secret-key")

def verify_api_key(x_api_key: Optional[str] = Header(None)):
    """Verify API key for authentication."""
    if x_api_key != VALID_API_KEY:
        raise HTTPException(
            status_code=401,
            detail=ErrorResponse(
                type="authentication_error",
                code="invalid_key",
                message="Invalid or missing API key",
                errors=[]
            ).dict()
        )

@router.post("/validate", response_model=MeasurementNormalized, dependencies=
[verify_api_key])
def validate_measurements(input_data: MeasurementInput):
    """
    Validate and normalize measurement input.

    If MediaPipe landmarks are provided, calculates measurements from
landmarks.
    Otherwise, uses user-provided measurements and converts to centimeters.

    Returns normalized measurements with confidence scores and accuracy
estimates.
    """
    try:
        normalized = normalize_and_validate(input_data)

        # Check accuracy threshold
        if normalized.accuracy_estimate and normalized.accuracy_estimate >
0.03:
            # Accuracy below 97% - flag for potential vendor calibration
            # TODO: Store flag in database for CEO Agent review
            pass

        return normalized

    except HTTPException:
        raise
    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=ErrorResponse(
                type="server_error",
                code="internal",
                message="An unexpected error occurred during validation",
                errors=[ErrorDetail(field="", message=str(e))],
                session_id=input_data.session_id
            ).dict()
        )

@router.post("/recommend", response_model=dict, dependencies=[verify_api_key])
def recommend_sizes(measurements: MeasurementNormalized):
    """
    Generate size recommendations from normalized measurements.

    Returns recommendations with confidence scores, processed measurements,
    and model version for API consumers.
    """
    try:
        m_dict = measurements.dict()
        recs = [recommend_top(m_dict), recommend_bottom(m_dict)]

        return {
            "recommendations": recs,
            "processed_measurements": measurements,
            "model_version": measurements.model_version,
            "session_id": measurements.session_id
        }

    except Exception as e:
        raise HTTPException(
            status_code=500,
            detail=ErrorResponse(
                type="server_error",
                code="internal",
                message="An unexpected error occurred during recommendation",
                errors=[ErrorDetail(field="", message=str(e))],
                session_id=measurements.session_id
            ).dict()
        )
```

3.5. Update Main Application

Update the main application to use the new router and include API metadata for external consumers.

File: `backend/app/main.py` (UPDATE)

```python
from fastapi import FastAPI
from backend.app.routers.measurements import router as measurements_router

app = FastAPI(
    title="FitTwin DMaaS API",
    description=(
        "Data-Model-as-a-Service API for accurate body measurements and size recommendations. "
        "Designed for AI systems (like ChatGPT-powered shopping assistants) and online retailers. "
        "Uses MediaPipe Pose Landmarker for free, on-device measurement extraction."
    ),
    version="1.0.0-mediapipe-mvp",
    contact={
        "name": "FitTwin Support",
        "email": "support@fittwin.com",
    },
    license_info={
        "name": "Proprietary",
    },
)

app.include_router(measurements_router)

@app.get("/")
def root():
    """Health check endpoint."""
    return {
        "status": "ok",
        "message": "FitTwin DMaaS API is running",
        "version": "1.0.0-mediapipe-mvp",
        "docs": "/docs"
    }

@app.get("/health")
def health():
    """Detailed health check for monitoring."""
    return {
        "status": "healthy",
        "database": "connected",  # TODO: Add actual DB health check
        "mediapipe": "available",
        "version": "1.0.0-mediapipe-mvp"
    }
```

---

4. MediaPipe Integration

4.1. MediaPipe Pose Landmarker Setup

The MediaPipe integration will be implemented in the Flutter/SwiftUI frontend app. This section provides guidance for the frontend implementation.

Key Requirements: - Use MediaPipe Pose Landmarker v3.1 (latest stable version) - Capture two photos: front-facing and side-facing - Detect landmarks on-device (no cloud processing) - Send landmarks to backend for measurement calculation - Store raw photos for future model training

Recommended Flutter Package:

```yaml
dependencies:
  google_ml_kit: ^0.16.0  # Includes MediaPipe Pose
  camera: ^0.10.0
  image_picker: ^0.8.0
```

Recommended iOS (SwiftUI) Framework:

```swift
import MediaPipeTasksVision
```

4.2. Landmark Detection Flow

1. **User opens app** and taps "Start Measurement"
2. **App requests camera permission** (if not already granted)
3. **App displays camera view** with on-screen guidance (e.g., "Stand 6 feet from camera, arms at sides")
4. **User captures front photo** - app detects landmarks in real-time and provides feedback
5. **User rotates 90 degrees** and captures side photo
6. **App processes both photos** with MediaPipe Pose Landmarker
7. **App sends data to backend** (photos + landmarks) via `/measurements/validate`
8. **Backend calculates measurements** and returns normalized data
9. **App displays results** and optionally calls `/measurements/recommend` for size recommendations

4.3. Landmark Quality Checks

The app should perform quality checks before sending data to the backend:

- **Visibility:** All key landmarks (shoulders, hips, ankles) must have visibility > 0.5
- **Symmetry:** Left and right side landmarks should be roughly symmetric
- **Pose:** User should be standing upright (not sitting, bending, or lying down)

- **Distance:** User should be 5-7 feet from camera (estimated from landmark scale)

If quality checks fail, the app should prompt the user to retake the photo.

---

5. Agent System Implementation

5.1. Agent Tools with Retry Logic

Implement the two tools with timeout, retry, and circuit breaker logic as recommended by ChatGPT.

File: `agents/tools/measurement_tools.py` (NEW)

```python
 import requests
import os
import time
from crewai import tool
from typing import Optional

# Configuration
API_BASE_URL = os.getenv("API_BASE_URL", "http://localhost:8000")
API_KEY = os.getenv("X_API_KEY", "staging-secret-key")
TIMEOUT = 10  # seconds
MAX_RETRIES = 1

class CircuitBreaker:
    """Simple circuit breaker to prevent thrashing on repeated failures."""
    def __init__(self, failure_threshold=3):
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.is_open = False

    def call_failed(self):
        self.failure_count += 1
        if self.failure_count >= self.failure_threshold:
            self.is_open = True

    def call_succeeded(self):
        self.failure_count = 0
        self.is_open = False

    def can_proceed(self):
        return not self.is_open

# Global circuit breakers for each endpoint
validate_breaker = CircuitBreaker()
recommend_breaker = CircuitBreaker()

@tool("validate_measurements")
def validate_measurements(measurement_data: dict) -> dict:
    """
    Validates and normalizes measurement data via the backend API.

    Supports both user-provided measurements and MediaPipe landmarks.
    Returns normalized measurements in centimeters with confidence scores.
    """
    if not validate_breaker.can_proceed():
        return {
            "error": "Circuit breaker is open. Too many recent failures.",
            "type": "circuit_breaker_error"
        }

    headers = {"X-API-Key": API_KEY, "Content-Type": "application/json"}
    url = f"{API_BASE_URL}/measurements/validate"

    for attempt in range(MAX_RETRIES + 1):
        try:
            response = requests.post(url, json=measurement_data,
headers=headers, timeout=TIMEOUT)

            if response.status_code == 200:
                validate_breaker.call_succeeded()
                return response.json()

            elif response.status_code == 422:
                # Validation error - don't retry, return error details
                validate_breaker.call_succeeded()  # Not a system failure
                error_data = response.json()
                return {
                    "error": error_data.get("detail", {}),
                    "status_code": 422
                }

            elif response.status_code in [500, 502, 503, 504]:
                # Server error - retry once
                if attempt < MAX_RETRIES:
                    time.sleep(2 ** attempt)  # Exponential backoff
                    continue
                else:
                    validate_breaker.call_failed()
                    return {
                        "error": f"Server error after {MAX_RETRIES + 1}
attempts",
                        "status_code": response.status_code,
                        "type": "server_error"
                    }

            elif response.status_code == 429:
                # Rate limit - retry once with backoff
                if attempt < MAX_RETRIES:
                    time.sleep(5)
                    continue
                else:
                    return {
                        "error": "Rate limit exceeded",
                        "status_code": 429,
                        "type": "rate_limit_error"
                    }

            else:
                # Other error
                validate_breaker.call_failed()
                return {
                    "error": f"Unexpected status code: {response.status_code}",
                    "status_code": response.status_code,
                    "type": "unexpected_error"
                }

        except requests.exceptions.Timeout:
            if attempt < MAX_RETRIES:
                time.sleep(2 ** attempt)
                continue
            else:
                validate_breaker.call_failed()
                return {
                    "error": "Request timed out",
                    "type": "timeout_error"
                }

        except requests.exceptions.RequestException as e:
            validate_breaker.call_failed()
            return {
                "error": f"Request failed: {str(e)}",
                "type": "connection_error"
            }

    return {"error": "Unexpected failure in retry loop", "type":
"unknown_error"}

@tool("recommend_sizes")
def recommend_sizes(normalized_measurements: dict) -> dict:
    """
    Generates size recommendations from normalized measurements.
    Returns recommendations with confidence scores, processed measurements,
    and model version for API consumers.
    """
    if not recommend_breaker.can_proceed():
        return {
            "error": "Circuit breaker is open. Too many recent failures.",
            "type": "circuit_breaker_error"
        }

    headers = {"X-API-Key": API_KEY, "Content-Type": "application/json"}
    url = f"{API_BASE_URL}/measurements/recommend"

    for attempt in range(MAX_RETRIES + 1):
        try:
            response = requests.post(url, json=normalized_measurements,
headers=headers, timeout=TIMEOUT)

            if response.status_code == 200:
                recommend_breaker.call_succeeded()
                return response.json()

            elif response.status_code in [500, 502, 503, 504]:
                if attempt < MAX_RETRIES:
                    time.sleep(2 ** attempt)
                    continue
                else:
                    recommend_breaker.call_failed()
                    return {
                        "error": f"Server error after {MAX_RETRIES + 1}
attempts",
                        "status_code": response.status_code,
                        "type": "server_error"
                    }

            elif response.status_code == 429:
                if attempt < MAX_RETRIES:
                    time.sleep(5)
                    continue
                else:
                    return {
                        "error": "Rate limit exceeded",
                        "status_code": 429,
                        "type": "rate_limit_error"
                    }

            else:
                recommend_breaker.call_failed()
                return {
                    "error": f"Unexpected status code: {response.status_code}",
                    "status_code": response.status_code,
                    "type": "unexpected_error"
                }

        except requests.exceptions.Timeout:
            if attempt < MAX_RETRIES:
                time.sleep(2 ** attempt)
                continue
            else:
                recommend_breaker.call_failed()
                return {
                    "error": "Request timed out",
                    "type": "timeout_error"
                }

        except requests.exceptions.RequestException as e:
            recommend_breaker.call_failed()
```

```
                return {
                    "error": f"Request failed: {str(e)}",
                    "type": "connection_error"
                }

        return {"error": "Unexpected failure in retry loop", "type":
"unknown_error"}
```

5.2. Agent Definitions with Strategic Directives

Define the five agents (CEO, Architect, ML Engineer, DevOps, Reviewer) with specific directives aligned to the DMaaS MVP strategy.

File: `agents/crew/measurement_crew.py` (NEW)

```python
import os
from crewai import Agent, Task, Crew, LLM
from agents.tools.measurement_tools import validate_measurements,
recommend_sizes

def create_measurement_crew():
    """Create and configure the measurement processing crew with strategic
directives."""

    # Ensure API key is provided
    api_key = os.getenv("OPENAI_API_KEY")
    if not api_key:
        raise RuntimeError("OPENAI_API_KEY not found in environment.")

    # Create LLM
    llm = LLM(model=os.getenv("AGENT_MODEL", "gpt-4o-mini"), api_key=api_key)

    # Define agents with strategic directives
    ceo = Agent(
        role="CEO",
        goal="Oversee MVP accuracy (<3% error) and Supabase integration within
5-day timeline and <$500 budget",
        backstory=(
            "You are the CEO of FitTwin, building a DMaaS platform for AI
systems and retailers. "
            "Your directive: Achieve a deployable Supabase-backed API with
MediaPipe-only measurement "
            "extraction within 5 days, with a budget constraint of <$500. "
            "You delegate tasks to the Architect, ML Engineer, and DevOps
agents. "
            "You communicate with the user when input is needed or when errors
cannot be resolved automatically. "
            "If MediaPipe accuracy falls below 97%, you trigger optional vendor
API calibration."
        ),
        llm=llm,
        verbose=True,
    )

    architect = Agent(
        role="Architect",
        goal="Implement Supabase schema and geometric equations for MediaPipe
measurement calculation",
        backstory=(
            "You are a meticulous architect who plans the data flow and system
design. "
            "Your directive: Prioritize Supabase and data security (RLS).
Design schema for measurement "
            "provenance (storing raw photos, MediaPipe landmarks, calculated
measurements). "
            "Implement geometric equations to calculate anthropometric
measurements from MediaPipe landmarks. "
            "You call validate_measurements first, then pass the normalized
result to recommend_sizes. "
            "If validation returns a 422 error with an obvious fix (like unit
conversion or name typo), "
            "you attempt ONE repair and retry. If the fix fails or the error is
complex, you escalate to the CEO."
        ),
        tools=[validate_measurements],
        llm=llm,
        verbose=True,
    )

    ml_engineer = Agent(
        role="ML Engineer",
        goal="Build internal model for measurement prediction and accuracy
estimation",
        backstory=(
            "You are an ML engineer who builds the core IP: the proprietary
measurement model. "
            "Your directive: Implement the data pipeline to ingest and
normalize MediaPipe landmarks and "
            "vendor measurements (if used for calibration). Design the initial
proprietary sizing model. "
            "You receive normalized measurements from the Architect and call
recommend_sizes. "
            "You return the JSON result with no extra storytelling. "
            "You also estimate measurement accuracy and flag low-confidence
results for CEO review."
        ),
        tools=[recommend_sizes],
        llm=llm,
        verbose=True,
    )

    devops = Agent(
        role="DevOps",
        goal="Manage CI/CD, TestFlight deployment, and infrastructure (<$500
budget)",
        backstory=(
            "You are a DevOps engineer who handles infrastructure, CI/CD, and
deployment. "
            "Your directive: Scaffold the Flutter/SwiftUI repo. Implement
GitHub Actions for linting, "
            "testing, and automated deployment. Deploy to TestFlight for user
testing. "
            "Provide final Supabase connection strings and DMaaS API keys. "
            "You work within a <$500 budget constraint, prioritizing free tiers
(Supabase, GitHub Actions, TestFlight)."
        ),
        llm=llm,
        verbose=True,
    )

    reviewer = Agent(
        role="Reviewer",
        goal="Validate cost and data security compliance before deployment",
        backstory=(
            "You are a code reviewer who critiques all code and infrastructure
changes before the DevOps Agent commits them. "
            "Your directive: Check for security vulnerabilities, code quality
issues, and alignment with the DMaaS strategy. "
            "Validate that cost constraints are met (<$500 for MVP). "
            "Ensure data security (RLS policies, API key authentication, secure
photo storage). "
            "Provide feedback to the Architect and ML Engineer for
improvements. "
            "This acts as an autonomous peer review to enhance security and
code quality."
        ),
        llm=llm,
        verbose=True,
    )

    # Define tasks
    validate_task = Task(
        description=(
            "Validate the user-provided measurement data or MediaPipe landmarks
using the validate_measurements tool. "
            "If you receive a 422 error with a clear repair hint (e.g., wrong
unit or typo in field name), "
            "attempt to fix it ONCE and retry. If the error persists or is
unclear, escalate to the CEO. "
            "If accuracy estimate is >3%, flag for potential vendor
calibration."
        ),
        agent=architect,
        expected_output="Normalized measurement data in centimeters with
confidence scores, or an error report for the CEO.",
    )

    recommend_task = Task(
        description=(
            "Use the normalized measurements from the Architect to generate
size recommendations. "
            "Call the recommend_sizes tool and return the result as JSON. "
            "Include confidence scores and model version for API consumers."
        ),
        agent=ml_engineer,
        expected_output="A JSON object with size recommendations, processed
measurements, and model version.",
    )

    review_task = Task(
        description=(
            "Review the complete workflow (validation + recommendation) for
security, cost, and quality. "
            "Check that RLS policies are in place, API keys are secure, and
budget is <$500. "
            "Provide feedback if any issues are found."
        ),
        agent=reviewer,
        expected_output="A review report with any issues found, or confirmation
that the system is ready for deployment.",
    )

    # Assemble crew
    crew = Crew(
        agents=[ceo, architect, ml_engineer, devops, reviewer],
        tasks=[validate_task, recommend_task, review_task],
        verbose=True,
    )

    return crew

def main():
    """Run the measurement crew with sample input."""
    crew = create_measurement_crew()

    # Example input with MediaPipe landmarks (placeholder)
    sample_input = {
        "front_landmarks": {
            "landmarks": [],  # TODO: Add actual landmark data
            "timestamp": "2025-10-26T15:00:00Z",
            "image_width": 1920,
            "image_height": 1080
        },
        "side_landmarks": {
            "landmarks": [],  # TODO: Add actual landmark data
            "timestamp": "2025-10-26T15:00:05Z",
            "image_width": 1920,
            "image_height": 1080
        },
        "front_photo_url":
"https://storage.fittwin.com/photos/session123/front.jpg",
        "side_photo_url":
```

```
"https://storage.fittwin.com/photos/session123/side.jpg",
        "session_id": "session123"
    }

    print("\n=== Starting Measurement Crew ===\n")
    print(f"Input: {sample_input}\n")

    result = crew.kickoff()

    print("\n=== Crew Output ===\n")
    print(result)

if __name__ == "__main__":
    main()
```

---

6. Database Schema and Data Provenance

6.1. Supabase Schema Design

Create tables for storing measurement provenance, including raw photos, MediaPipe landmarks, calculated measurements, and size recommendations.

File: `data/supabase/migrations/002_measurement_provenance.sql` (NEW)

```sql
-- Measurement sessions table
CREATE TABLE measurement_sessions (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    user_id UUID REFERENCES auth.users(id),
    session_id TEXT UNIQUE NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    updated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    status TEXT DEFAULT 'pending', -- pending, completed, failed
    accuracy_estimate FLOAT,
    needs_calibration BOOLEAN DEFAULT FALSE
);

-- Raw photos table
CREATE TABLE measurement_photos (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES measurement_sessions(id) ON DELETE CASCADE,
    photo_type TEXT NOT NULL, -- front, side
    photo_url TEXT NOT NULL,
    uploaded_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    image_width INT,
    image_height INT
);

-- MediaPipe landmarks table
CREATE TABLE mediapipe_landmarks (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES measurement_sessions(id) ON DELETE CASCADE,
    photo_id UUID REFERENCES measurement_photos(id) ON DELETE CASCADE,
    landmark_type TEXT NOT NULL, -- front, side
    landmarks JSONB NOT NULL, -- Array of {x, y, z, visibility}
    detected_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    model_version TEXT DEFAULT 'v3.1'
);

-- Calculated measurements table (MediaPipe-derived)
CREATE TABLE measurements_mediapipe (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES measurement_sessions(id) ON DELETE CASCADE,
    height_cm FLOAT NOT NULL,
    neck_cm FLOAT,
    shoulder_cm FLOAT,
    chest_cm FLOAT,
    underbust_cm FLOAT,
    waist_natural_cm FLOAT,
    sleeve_cm FLOAT,
    bicep_cm FLOAT,
    forearm_cm FLOAT,
    hip_low_cm FLOAT,
    thigh_cm FLOAT,
    knee_cm FLOAT,
    calf_cm FLOAT,
    ankle_cm FLOAT,
    front_rise_cm FLOAT,
    back_rise_cm FLOAT,
    inseam_cm FLOAT,
    outseam_cm FLOAT,
    confidence FLOAT DEFAULT 1.0,
    accuracy_estimate FLOAT,
    model_version TEXT DEFAULT 'v1.0-mediapipe',
    calculated_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Vendor measurements table (for calibration only, if needed)
CREATE TABLE measurements_vendor (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES measurement_sessions(id) ON DELETE CASCADE,
    vendor_name TEXT NOT NULL, -- 3dlook, nettelo, etc.
    vendor_version TEXT,
    measurements JSONB NOT NULL, -- Raw vendor JSON response
    confidence FLOAT,
    cost_usd FLOAT,
    called_at TIMESTAMP WITH TIME ZONE DEFAULT NOW(),
    used_for_calibration BOOLEAN DEFAULT TRUE,
    excluded_from_live BOOLEAN DEFAULT TRUE -- Never use in production
);

-- Size recommendations table
CREATE TABLE size_recommendations (
    id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    session_id UUID REFERENCES measurement_sessions(id) ON DELETE CASCADE,
    measurement_id UUID REFERENCES measurements_mediapipe(id) ON DELETE
CASCADE,
    category TEXT NOT NULL, -- tops, bottoms, dresses, etc.
    size TEXT NOT NULL,
    confidence FLOAT NOT NULL,
    rationale TEXT,
    model_version TEXT DEFAULT 'v1.0',
    created_at TIMESTAMP WITH TIME ZONE DEFAULT NOW()
);

-- Indexes for performance
CREATE INDEX idx_sessions_user ON measurement_sessions(user_id);
CREATE INDEX idx_sessions_session_id ON measurement_sessions(session_id);
CREATE INDEX idx_photos_session ON measurement_photos(session_id);
CREATE INDEX idx_landmarks_session ON mediapipe_landmarks(session_id);
CREATE INDEX idx_measurements_session ON measurements_mediapipe(session_id);
CREATE INDEX idx_vendor_session ON measurements_vendor(session_id);
CREATE INDEX idx_recommendations_session ON size_recommendations(session_id);

-- Row Level Security (RLS) policies
ALTER TABLE measurement_sessions ENABLE ROW LEVEL SECURITY;
ALTER TABLE measurement_photos ENABLE ROW LEVEL SECURITY;
ALTER TABLE mediapipe_landmarks ENABLE ROW LEVEL SECURITY;
ALTER TABLE measurements_mediapipe ENABLE ROW LEVEL SECURITY;
ALTER TABLE measurements_vendor ENABLE ROW LEVEL SECURITY;
ALTER TABLE size_recommendations ENABLE ROW LEVEL SECURITY;

-- Users can only access their own data
CREATE POLICY "Users can view own sessions" ON measurement_sessions
    FOR SELECT USING (auth.uid() = user_id);

CREATE POLICY "Users can insert own sessions" ON measurement_sessions
    FOR INSERT WITH CHECK (auth.uid() = user_id);

CREATE POLICY "Users can view own photos" ON measurement_photos
    FOR SELECT USING (
        session_id IN (SELECT id FROM measurement_sessions WHERE user_id =
auth.uid())
    );

CREATE POLICY "Users can insert own photos" ON measurement_photos
    FOR INSERT WITH CHECK (
        session_id IN (SELECT id FROM measurement_sessions WHERE user_id =
auth.uid())
    );

-- Similar policies for other tables...
-- (Add policies for landmarks, measurements, vendor, recommendations)

-- API service role can access all data (for DMaaS API)
CREATE POLICY "Service role full access sessions" ON measurement_sessions
```

```
    FOR ALL USING (auth.jwt()->>'role' = 'service_role');

CREATE POLICY "Service role full access photos" ON measurement_photos
    FOR ALL USING (auth.jwt()->>'role' = 'service_role');

-- Similar service role policies for other tables...
```

6.2. Data Storage Flow

When the backend receives a measurement validation request:

1. **Create session record** in `measurement_sessions` table
2. **Store raw photos** in Supabase Storage and create records in `measurement_photos`
3. **Store MediaPipe landmarks** in `mediapipe_landmarks` table
4. **Calculate measurements** using geometric equations
5. **Store calculated measurements** in `measurements_mediapipe` table
6. **Check accuracy estimate** - if >3% error, set `needs_calibration = TRUE`
7. **Return normalized measurements** to client

When the backend receives a recommendation request:

1. **Retrieve measurements** from `measurements_mediapipe` table
2. **Apply fit rules** to generate size recommendations
3. **Store recommendations** in `size_recommendations` table
4. **Return recommendations** to client

If calibration is needed (accuracy <97%):

1. **CEO Agent reviews** sessions with `needs_calibration = TRUE`
2. **CEO Agent triggers** vendor API call for 50-100 sessions
3. **Store vendor measurements** in `measurements_vendor` table with `used_for_calibration = TRUE`
4. **ML Engineer Agent** uses calibration data to refine geometric equations
5. **Deploy updated equations** to production
6. **Exclude vendor data** from live systems (`excluded_from_live = TRUE`)

7. Testing and Validation Strategy

7.1. Backend Unit Tests

Create comprehensive tests for the new endpoints and validation logic.

File: `tests/backend/test_validation_endpoint.py` (NEW)

```python
from fastapi.testclient import TestClient
import sys
from pathlib import Path

project_root = Path(__file__).resolve().parents[2]
sys.path.insert(0, str(project_root))

from backend.app.main import app

client = TestClient(app)

def test_validate_golden_payload_user_input():
    """Test with a valid user-provided payload - should return normalized
measurements."""
    payload = {
        "waist_natural": 32,
        "hip_low": 40,
        "inseam": 30,
        "unit": "in",
        "session_id": "test-session-1"
    }
    headers = {"X-API-Key": "staging-secret-key"}

    response = client.post("/measurements/validate", json=payload,
headers=headers)

    assert response.status_code == 200
    data = response.json()
    assert "waist_natural_cm" in data
    assert data["waist_natural_cm"] == 32 * 2.54  # Converted to cm
    assert data["source"] == "user_input"
    assert data["session_id"] == "test-session-1"

def test_validate_golden_payload_mediapipe():
    """Test with valid MediaPipe landmarks - should return calculated
measurements."""
    payload = {
        "front_landmarks": {
            "landmarks": [{"x": 0.5, "y": 0.5, "z": 0.0, "visibility": 0.9}] *
33,  # 33 landmarks
            "timestamp": "2025-10-26T15:00:00Z",
            "image_width": 1920,
            "image_height": 1080
        },
        "side_landmarks": {
            "landmarks": [{"x": 0.5, "y": 0.5, "z": 0.0, "visibility": 0.9}] *
33,
            "timestamp": "2025-10-26T15:00:05Z",
            "image_width": 1920,
            "image_height": 1080
        },
        "front_photo_url": "https://storage.fittwin.com/photos/test/front.jpg",
        "side_photo_url": "https://storage.fittwin.com/photos/test/side.jpg",
        "session_id": "test-session-2"
    }
    headers = {"X-API-Key": "staging-secret-key"}

    response = client.post("/measurements/validate", json=payload,
headers=headers)

    assert response.status_code == 200
    data = response.json()
    assert "height_cm" in data
    assert data["source"] == "mediapipe"
    assert data["model_version"] == "v1.0-mediapipe"
    assert data["session_id"] == "test-session-2"
    assert "accuracy_estimate" in data

def test_validate_broken_payload():
    """Test with an invalid payload - should return 422 with error details."""
    payload = {
        "waist_circ": 32,  # Wrong field name
        "unit": "in"
    }
    headers = {"X-API-Key": "staging-secret-key"}

    response = client.post("/measurements/validate", json=payload,
headers=headers)

    assert response.status_code == 422
    error = response.json()["detail"]
    assert error["type"] == "validation_error"
    assert error["code"] == "unknown_field"
    assert len(error["errors"]) > 0
    assert "waist_circ" in error["errors"][0]["field"]
    assert "hint" in error["errors"][0]

def test_validate_missing_api_key():
    """Test without API key - should return 401."""
    payload = {"waist_natural": 32, "unit": "cm"}

    response = client.post("/measurements/validate", json=payload)

    assert response.status_code == 401
    error = response.json()["detail"]
    assert error["type"] == "authentication_error"

def test_recommend_golden_payload():
    """Test recommendation with valid normalized measurements."""
    payload = {
        "height_cm": 170.0,
        "neck_cm": 40,
        "shoulder_cm": 45,
        "chest_cm": 100,
        "underbust_cm": 85,
        "waist_natural_cm": 80,
        "sleeve_cm": 60,
        "bicep_cm": 30,
        "forearm_cm": 25,
        "hip_low_cm": 100,
        "thigh_cm": 55,
        "knee_cm": 38,
        "calf_cm": 35,
        "ankle_cm": 22,
        "front_rise_cm": 25,
        "back_rise_cm": 35,
        "inseam_cm": 76,
        "outseam_cm": 100,
        "source": "mediapipe",
        "model_version": "v1.0-mediapipe",
        "confidence": 0.95,
        "session_id": "test-session-3"
    }
    headers = {"X-API-Key": "staging-secret-key"}

    response = client.post("/measurements/recommend", json=payload,
headers=headers)

    assert response.status_code == 200
    data = response.json()
    assert "recommendations" in data
    assert "model_version" in data
    assert "session_id" in data
    assert len(data["recommendations"]) > 0
    assert data["session_id"] == "test-session-3"
```

7.2. Agent Tool Tests

Create tests for the agent tools with mocked API responses.

File: `tests/agents/test_measurement_tools.py` (NEW)

```python
 import pytest
from unittest.mock import patch, MagicMock
import sys
from pathlib import Path

project_root = Path(__file__).resolve().parents[2]
sys.path.insert(0, str(project_root))

from agents.tools.measurement_tools import validate_measurements,
recommend_sizes

def test_validate_measurements_success():
    """Test successful validation with MediaPipe landmarks."""
    with patch(\'agents.tools.measurement_tools.requests.post\') as mock_post:
        mock_response = MagicMock()
        mock_response.status_code = 200
        mock_response.json.return_value = {
            "height_cm": 170.0,
            "waist_natural_cm": 81.28,
            "source": "mediapipe",
            "confidence": 0.95,
            "accuracy_estimate": 0.05
        }
        mock_post.return_value = mock_response

        result = validate_measurements({
            "front_landmarks": {"landmarks": [], "timestamp": "2025-10-
26T15:00:00Z"},
            "side_landmarks": {"landmarks": [], "timestamp": "2025-10-
26T15:00:05Z"}
        })

        assert "height_cm" in result
        assert result["source"] == "mediapipe"
        assert result["confidence"] == 0.95

def test_validate_measurements_422_error():
    """Test validation error handling."""
    with patch(\'agents.tools.measurement_tools.requests.post\') as mock_post:
        mock_response = MagicMock()
        mock_response.status_code = 422
        mock_response.json.return_value = {
            "detail": {
                "type": "validation_error",
                "code": "unknown_field",
                "message": "Unknown field",
                "errors": [{"field": "waist_circ", "message": "Unknown field",
"hint": "Did you mean waist_natural?"}]
            }
        }
        mock_post.return_value = mock_response

        result = validate_measurements({"waist_circ": 32, "unit": "in"})

        assert "error" in result
        assert result["status_code"] == 422
        assert "waist_circ" in str(result["error"])

def test_validate_measurements_timeout():
    """Test timeout handling with retry."""
    with patch(\'agents.tools.measurement_tools.requests.post\') as mock_post:
        import requests
        mock_post.side_effect = requests.exceptions.Timeout()

        result = validate_measurements({"waist_natural": 32, "unit": "in"})
        assert "error" in result
        assert result["type"] == "timeout_error"
        assert mock_post.call_count == 2  # Initial + 1 retry

def test_recommend_sizes_success():
    """Test successful size recommendation."""
    with patch(\'agents.tools.measurement_tools.requests.post\') as mock_post:
        mock_response = MagicMock()
        mock_response.status_code = 200
        mock_response.json.return_value = {
            "recommendations": [
                {"category": "tops", "size": "M", "confidence": 0.9,
"rationale": "Based on chest and waist"},
                {"category": "bottoms", "size": "32", "confidence": 0.85,
"rationale": "Based on waist and inseam"}
            ],
            "model_version": "v1.0-mediapipe",
            "session_id": "test-123"
        }
        mock_post.return_value = mock_response

        result = recommend_sizes({
            "height_cm": 170.0,
            "chest_cm": 100,
            "waist_natural_cm": 80,
            "source": "mediapipe"
        })

        assert "recommendations" in result
        assert len(result["recommendations"]) == 2
        assert result["recommendations"][0]["category"] == "tops"
```

7.3. Accuracy Validation with Tape Measure Benchmarks

To validate the MediaPipe measurement accuracy, conduct tape measure benchmarks:

1. **Recruit 20-30 test subjects** with diverse body types
2. **Measure each subject** manually with a tape measure (ground truth)
3. **Capture photos** and process with MediaPipe
4. **Compare MediaPipe measurements** to ground truth
5. **Calculate error percentage** for each dimension
6. **Target:** <3% error for core dimensions (height, chest, waist, hip, inseam)

If accuracy is <97%, trigger vendor API calibration: - Purchase 50-100 scans from 3DLOOK or Nettelo ($50-200 total) - Use vendor measurements to calibrate geometric equations - Re-test accuracy with calibrated equations - Deploy to production once accuracy ≥97%

8. Deployment and Infrastructure

8.1. Infrastructure Components

Component	Service	Cost	Notes
Database	Supabase (Free tier)	$0/month	500MB storage, 2GB bandwidth
API Hosting	Supabase Edge Functions or Vercel	$0-20/month	Free tier sufficient for MVP
Photo Storage	Supabase Storage	$0/month	1GB free
CI/CD	GitHub Actions	$0/month	Free for public repos
iOS App	TestFlight	$0/month	Free for testing
Monitoring	Supabase Dashboard	$0/month	Built-in

Total MVP Infrastructure Cost: <$20/month(likely$0 with free tiers)

8.2. GitHub Actions CI/CD Pipeline

File: `.github/workflows/backend-ci.yml` (NEW)

```yaml
 name: Backend CI

on:
  push:
    branches: [ main, develop ]
  pull_request:
    branches: [ main ]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - uses: actions/checkout@v3

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.11'

    - name: Install dependencies
      run: |
        python -m pip install --upgrade pip
        pip install -r backend/requirements.txt
        pip install pytest pytest-cov

    - name: Run tests
      run: |
        export PYTHONPATH="$`{PYTHONPATH}:`$(pwd)"
        pytest tests/backend/ -v --cov=backend --cov-report=term-missing

    - name: Lint with flake8
      run: |
        pip install flake8
        flake8 backend/ --count --select=E9,F63,F7,F82 --show-source --
statistics

  deploy:
    needs: test
    runs-on: ubuntu-latest
    if: github.ref == 'refs/heads/main'

    steps:
    - uses: actions/checkout@v3

    - name: Deploy to Supabase
      run: |
        # TODO: Add Supabase deployment commands
        echo "Deploying to Supabase..."
```

8.3. Deployment Checklist

Before deploying to production:

- [ ] All tests passing (backend unit tests, agent tool tests)
- [ ] Accuracy validation complete (⩾97% for core dimensions)
- [ ] Supabase schema deployed with RLS policies

- [ ] API key authentication configured
- [ ] Photo storage configured and tested
- [ ] CI/CD pipeline configured and tested
- [ ] TestFlight build deployed and tested
- [ ] API documentation generated (Swagger/OpenAPI)
- [ ] Monitoring and logging configured
- [ ] Cost analysis confirmed (<$500 for MVP)

---

9. Five-Day Implementation Timeline

Day 1: Architecture & Data Foundation

CEO Agent: - Finalize structured JSON brief for MVP - Set budget constraint (<$500) - Set accuracy target (<3% error)

Architect Agent: - Design Supabase schema with RLS and measurement provenance - Write migration scripts - Design geometric equations for MediaPipe measurement calculation

DevOps Agent: - Initialize Supabase project - Scaffold Flutter/SwiftUI repo - Set up GitHub Actions CI/CD pipeline

Deliverables: - Supabase schema deployed - Empty repo with CI/CD pipelines - Geometric equations documented

Day 2: Core Measurement Logic

ML Engineer Agent: - Implement MediaPipe pose estimation for on-device guidance (Flutter/SwiftUI) - Implement geometric equations in backend validation logic - Add accuracy estimation function

Architect Agent: - Implement backend validation endpoint (`/measurements/validate`) - Integrate MediaPipe landmark processing - Add unit tests for validation logic

Deliverables: - Working app with real-time pose guidance - Backend validation endpoint functional - Unit tests passing

Day 3: End-to-End Data Flow

DevOps Agent: - Implement photo upload to Supabase Storage - Create API endpoint to receive and store measurement data - Set up database triggers for provenance tracking

ML Engineer Agent: - Finalize data normalization and storage logic - Implement recommendation endpoint (`/measurements/recommend`) - Add fit rules for tops and bottoms

Deliverables: - End-to-end flow: App captures photo → Calls backend → Stores in Supabase - Recommendation endpoint functional

Day 4: API & Security

Architect Agent: - Define the FitTwin DMaaS API documentation (Swagger/OpenAPI) - Add API versioning and rate limiting - Implement vendor API flag for optional calibration

DevOps Agent: - Implement RLS policies on Supabase to secure data - Configure API key authentication - Test security with penetration testing tools

Reviewer Agent: - Full code and security review - Validate cost constraints (<$500) - Check data security compliance

Deliverables: - DMaaS API documentation (Markdown/Swagger) - Secure data storage verified - Security review complete

Day 5: Testing & Deployment

ML Engineer Agent: - Conduct tape-measure benchmarks with test subjects - Calculate accuracy for core dimensions - Flag sessions needing calibration if accuracy <97%

DevOps Agent: - Deploy to TestFlight for user testing - Provide final Supabase connection string and DMaaS API key - Set up monitoring and logging

CEO Agent: - Review accuracy results - Decide on vendor API calibration (if needed) - Prepare final report for stakeholders

Deliverables: - Deployable MVP (TestFlight link) - Accuracy validation report - Final documentation and access credentials

---

10. Cost Analysis and Efficiency

10.1. MVP Cost Breakdown

Item	Service/Vendor	Cost	Notes
MediaPipe extraction	Google MediaPipe (open source)	$0.00 per user	Free, on-device
Database	Supabase (Free tier)	$0/month	500MB storage, sufficient for MVP
Photo storage	Supabase Storage	$0/month	1GB free, ~200 sessions
API hosting	Supabase Edge Functions	$0/month	Free tier
CI/CD	GitHub Actions	$0/month	Free for public repos
iOS testing	TestFlight	$0/month	Free
Optional calibration	3DLOOK or Nettelo	$50 - 200$	Only if accuracy <97%
Monitoring	Supabase Dashboard	$0/month	Built-in

Total MVP Cost: $0 - 200$ (depending on calibration need)

Ongoing Cost (post-MVP): <$20/month (if staying on free tiers)

10.2. Cost Comparison: MediaPipe vs Vendor-Only

Approach	Per-Scan Cost	1,000 Users	10,000 Users	Data Ownership
MediaPipe-only (MVP)	$0.00|$0	$0	Full	
3DLOOK	$4.99|$4,990	$49,900	Partial	
Nettelo	$2.00|$2,000	$20,000	Partial	
Hybrid (MediaPipe + calibration)	$0.00+$50-200 one-time	$50 - 200|$50-200	Full	

Savings with MediaPipe-only approach: - At 1,000 users: $2,000−$4,990 saved - At 10,000 users: $20,000−$49,900 saved - At 100,000 users: $200,000−$499,000 saved

10.3. Cost-Efficiency Principles

1. **Free-first:** Prioritize free, open-source technologies (MediaPipe, Supabase free tier, GitHub Actions)
2. **Pay-per-value:** Only pay for services that directly improve accuracy or user experience
3. **Calibration-only vendor use:** Use vendor APIs only for one-time calibration, not per-scan
4. **Data ownership:** Build proprietary dataset to eliminate vendor dependencies over time
5. **Scalable free tiers:** Design system to stay within free tiers as long as possible

---

11. Next Steps and Milestones

11.1. Immediate Next Steps (This Week)

1. **Confirm this implementation plan** with stakeholders
2. **Set up development environment** (Supabase project, GitHub repo, local dev setup)

3. **Implement Day 1 tasks** (schema design, repo scaffolding, CI/CD setup)
4. **Begin MediaPipe integration** in Flutter/SwiftUI app
5. **Implement backend validation endpoint** with geometric equations

11.2. Week 1 Milestones

- [ ] Supabase schema deployed with RLS policies
- [ ] Backend validation endpoint functional
- [ ] MediaPipe integration in app (landmark detection)
- [ ] Unit tests passing for backend
- [ ] CI/CD pipeline configured

11.3. Week 2 Milestones

- [ ] End-to-end data flow working (app → backend → database)
- [ ] Recommendation endpoint functional
- [ ] API documentation complete
- [ ] Security review complete
- [ ] TestFlight build deployed

11.4. Week 3 Milestones

- [ ] Tape-measure benchmarks complete
- [ ] Accuracy validation report ready
- [ ] Decision on vendor calibration (if needed)
- [ ] Final MVP deployed to TestFlight
- [ ] DMaaS API ready for external customers

11.5. Long-Term Roadmap (Post-MVP)

Month 2-3: Calibration and Refinement - Conduct vendor API calibration if accuracy <97% - Refine geometric equations based on calibration data - Expand test subject pool to 100+ users - Improve accuracy to >98%

Month 4-6: DMaaS Launch - Onboard first AI/retailer customers - Create API integration guides and SDKs - Implement rate limiting and tiered pricing - Scale infrastructure beyond free tiers

Month 7-12: Proprietary Model Training - Collect 10,000+ measurement sessions - Train proprietary ML model for size recommendations - Phase out vendor APIs entirely - Achieve >99% accuracy with proprietary model

---

12. Three-Mode Universal Architecture (v2.0)

This updated architecture expands the FitTwin DMaaS platform to support three modes of application, ensuring 100% market coverage across all devices.

12.1. Platform Coverage Matrix

Platform	Method	Accuracy	Cost/Scan	Coverage
iOS (12 Pro+)	ARKit + LiDAR	~99%	$0	~40% of iOS users
iOS (older)	MediaPipe (Native)	~95%	$0	~60% of iOS users
Android	MediaPipe (Native)	~95%	$0	100% of Android
Web (modern)	MediaPipe Web	92-95%	$0	~80% of web users
Web (legacy)	Server MediaPipe	92-95%	~$0.01	~20% of web users
Desktop	MediaPipe Web	90-92%	$0	100% of desktop

Total Coverage: 100% of all users across all platforms.

12.2. Updated Backend API Design

The backend API will be updated to accept measurements from all three sources.

File: `backend/app/schemas/measure_schema.py` (UPDATED)

```
class MeasurementInput(BaseModel):
    # Source type
    source_type: str  # "arkit_lidar", "mediapipe_native", "mediapipe_web",
"user_input"
    platform: str     # "ios", "android", "web_mobile", "web_desktop"

    # ARKit LiDAR data (iOS native only)
    arkit_body_anchor: Optional[dict] = None
    arkit_depth_map: Optional[str] = None

    # MediaPipe data (native apps + web)
    front_landmarks: Optional[MediaPipeLandmarks] = None
    side_landmarks: Optional[MediaPipeLandmarks] = None

    # Web-specific metadata
    browser_info: Optional[dict] = None
    processing_location: Optional[str] = None  # "client", "server"

    # Photos for provenance
    front_photo_url: Optional[str] = None
    side_photo_url: Optional[str] = None

    # Session metadata
    session_id: Optional[str] = None
    device_id: Optional[str] = None
```

12.3. Web App Architecture

A new web application will be created to support browser-based measurements.

Directory: `web-app/` (NEW)

```
web-app/
├── pages/
│   ├── index.tsx              # Landing page
│   ├── capture.tsx            # Photo capture flow
│   └── results.tsx            # Measurement results
├── components/
│   ├── CameraCapture.tsx      # WebRTC camera access
│   ├── PoseGuide.tsx          # Real-time pose guidance
│   └── MeasurementDisplay.tsx # Results visualization
├── lib/
│   ├── mediapipe-web.ts       # MediaPipe integration
│   ├── api-client.ts          # Backend API calls
│   └── storage.ts             # Local storage for sessions
└── public/
    └── models/                # MediaPipe WASM models
```

12.4. Updated Five-Day MVP Timeline

- **Day 1:** Architecture & Multi-Platform Design
- **Day 2:** iOS Native (LiDAR + MediaPipe)
- **Day 3:** Web App (MediaPipe Web)

- **Day 4:** Android Native + Backend Integration
- **Day 5:** Testing & Deployment

13. Web App User Experience Enhancements

13.1. Photo Capture Flow with Countdown

The web app implements a guided photo capture flow with 10-second countdown timers to ensure users have adequate time to position themselves correctly.

Front Photo Capture Flow

1. **Setup Screen**: Display positioning instructions
2. Stand 6-8 feet from camera
3. Press legs out, feet shoulder-width apart
4. Place arms at 30-45° from body
5. Face directly toward camera
6. Ensure good lighting and full body in frame
7. **Countdown Phase**: 10-second animated countdown
8. Large animated countdown display (10, 9, 8...)
9. Real-time positioning reminders
10. Visual guide showing correct pose
11. Automatic capture at 0 seconds
12. **Confirmation**: Photo captured successfully
13. Success message with checkmark
14. Preview of captured photo (optional)
15. Continue button to proceed

Side Photo Capture Flow

1. **Rotation Instruction**: Clear guidance to turn 90° right

2. "Turn 90° to your right (your right side faces the camera)"
3. Visual diagram showing rotation direction
4. **Setup Screen**: Side-specific positioning instructions
5. Stand with feet together or slightly apart
6. Place arms relaxed at sides
7. Stand up straight with good posture
8. Keep full body in frame from head to toe
9. **Countdown Phase**: 10-second animated countdown
10. Large animated countdown display
11. Side-specific positioning reminders
12. Visual guide showing correct side pose
13. Automatic capture at 0 seconds
14. **Processing**: Automatic transition to measurement processing

13.2. User Experience Benefits

Reduced User Error: - Countdown gives users time to adjust position - Clear instructions reduce need for retakes - Automatic capture eliminates timing issues

Improved Accuracy: - Proper positioning leads to better landmark detection - Consistent pose across users improves model performance - Reduced motion blur from manual capture

Professional Feel: - Countdown creates anticipation and engagement - Automated process feels more sophisticated - Visual guides provide confidence

13.3. Technical Implementation

State Management:

```
type CaptureStep =
  | "front-ready"
  | "front-countdown"
  | "front-captured"
  | "side-ready"
  | "side-countdown"
  | "side-captured"
  | "processing"
  | "complete";
```

Countdown Timer: - React `useEffect` hook for countdown logic - 1-second intervals using `setTimeout` - Automatic photo capture when countdown reaches 0 - Reset countdown between front and side captures

Progress Tracking: - Progress bar shows completion percentage - Updates at each step of the flow - Visual feedback for user engagement

13.4. Future Enhancements

- Real-time pose validation using MediaPipe landmarks
- Audio countdown announcements for accessibility
- Adjustable countdown duration (5, 10, 15 seconds)
- Pose quality feedback before capture
- Option to skip countdown for experienced users