Demeter - AI Engineer PRD

ML Ensemble & Simulation Engine

Role: AI/ML Engineer
Stack: Python 3.11 + FastAPI + scikit-learn + TensorFlow/Keras

---

1. Overview

Build the ML prediction service that combines physics-based models with machine learning to predict crop stress and simulate farming scenarios.

---

2. ML Ensemble Architecture

```
+--------------------------------------------------------------------+
|                       ML ENSEMBLE                                  |
+--------------------------------------------------------------------+
|                                                                    |
|  INPUT DATA                                                        |
|  +------------------------------------------------------------+   |
|  | soil_moisture, temperature, humidity, rainfall,            |   |
|  | growth_stage, days_since_rain, weather_forecast            |   |
|  +------------------------------------------------------------+   |
|                             |                                      |
|         +--------------------+--------------------+                |
|         v                    v                    v                |
|  +-----------------+   +-----------------+   +-----------------+   |
|  | PHYSICS MODEL   |   | RANDOM FOREST   |   |      LSTM       |   |
|  | Water Balance   |   |  Classifier     |   |  Time Series    |   |
|  |                 |   |                 |   |                 |   |
|  | - Hargreaves    |   | - 7 features    |   | - 14-day        |   |
|  |   ET0           |   | - 5 classes     |   |   window        |   |
|  | - FAO-56 Kc     |   |                 |   | - 7-day         |   |
|  | - Soil water    |   |                 |   |   forecast      |   |
|  |   deficit       |   |                 |   |                 |   |
|  |                 |   |                 |   |                 |   |
|  | Weight: 0.4     |   | Weight: 0.35    |   | Weight: 0.25    |   |
|  +-------+--------+   +-------+--------+   +-------+--------+   |
|          |                    |                    |               |
|          +--------------------+--------------------+               |
|                               |                                    |
|                               v                                    |
|                     +--------------------+                         |
|                     |   META-LEARNER     |                         |
|                     |  Weighted Average  |                         |
|                     |  + Confidence      |                         |
|                     +----------+---------+                         |
|                               |                                    |
|                               v                                    |
|                     +--------------------+                         |
|                     |       OUTPUT       |                         |
|                     | - stress_index     |                         |
|                     | - risk_category    |                         |
|                     | - confidence       |                         |
|                     | - days_to_critical |                         |
```

```
|                     | - recommendation   |                        |
|                     | - forecast[]       |                        |
|                     +--------------------+                        |
|                                                                   |
+-------------------------------------------------------------------+
```

---

3. API Endpoints

3.1 Prediction Endpoint

`POST /predict`

Request:

```json
{
  "farm_id": "uuid",
  "current_data": {
    "soil_moisture": 32.5,
    "temperature": 34.0,
    "humidity": 45.0,
    "days_since_rain": 5
  },
  "farm_info": {
    "planting_date": "2026-02-01",
    "growth_stage": "VEGETATIVE",
    "latitude": 10.5105,
    "longitude": 7.4165
  },
  "weather_forecast": [
    {"date": "2026-02-26", "temp_max": 36, "temp_min": 22, "precipitation": 0},
    {"date": "2026-02-27", "temp_max": 37, "temp_min": 23, "precipitation": 0}
  ],
  "historical_data": [
    {"timestamp": "2026-02-24T10:00:00Z", "soil_moisture": 45, "temperature": 32},
    {"timestamp": "2026-02-25T10:00:00Z", "soil_moisture": 38, "temperature": 33}
  ]
}
```

Response:

```json
{
  "stress_index": 72,
  "risk_category": "SEVERE",
  "confidence": 0.85,
  "days_to_critical": 4,
  "recommendation": "Irrigate 25mm within 3 days to prevent yield loss",
  "forecast": [
    {"day": 1, "stress": 72, "category": "SEVERE"},
    {"day": 2, "stress": 76, "category": "SEVERE"},
    {"day": 3, "stress": 79, "category": "SEVERE"},
    {"day": 4, "stress": 83, "category": "CRITICAL"},
    {"day": 5, "stress": 85, "category": "CRITICAL"},
    {"day": 6, "stress": 88, "category": "CRITICAL"},
    {"day": 7, "stress": 90, "category": "CRITICAL"}
  ],
  "model_contributions": {
    "physics": 0.4,
    "random_forest": 0.35,
```

```
    "lstm": 0.25
  }
}
```

3.2 Simulation Endpoint

```
POST /simulate
```

Request:


```json
{
  "farm_id": "uuid",
  "current_state": {
    "soil_moisture": 32.5,
    "stress_index": 45,
    "growth_stage": "VEGETATIVE",
    "days_since_planting": 30
  },
  "scenario": "DRY_WEEK",
  "parameters": {
    "duration_days": 14,
    "rainfall_mm": 0
  },
  "weather_forecast": [...]
}
```


Response:


```json
{
  "baseline": {
    "stress_index": 45,
    "yield_impact": 0,
    "trajectory": [45, 46, 47, 48, 49, 50, 51]
  },
  "simulated": {
    "stress_index": 82,
    "yield_impact": -35,
    "trajectory": [45, 52, 58, 64, 70, 76, 82]
  },
  "recommendation": "Irrigate 25mm before day 3 to avoid critical stress",
  "risk_window": {
    "critical_day": 4,
    "action_deadline": 3
  }
}
```


3.3 Health Check

```
GET /health
```

---

4. Model Implementations

4.1 Physics Model: Water Balance

```python
# models/water_balance.py

import numpy as np
from datetime import datetime, timedelta
```

```python
class WaterBalanceModel:
    """
    FAO-56 based water balance model for crop stress estimation.
    """

    # Crop coefficients (Kc) by growth stage for maize
    CROP_COEFFICIENTS = {
        'EMERGENCE': 0.3,
        'VEGETATIVE': 0.7,
        'TASSELING': 1.2,
        'GRAIN_FILL': 1.0,
        'MATURITY': 0.6
    }

    # Soil parameters (sandy loam - common in Northern Nigeria)
    FIELD_CAPACITY = 0.28  # m³/m³
    WILTING_POINT = 0.10   # m³/m³
    ROOT_DEPTH = 0.6       # meters

    def calculate_et0(self, temp_max: float, temp_min: float,
                      latitude: float, day_of_year: int) -> float:
        """
        Hargreaves equation for reference evapotranspiration.
        ET0 = 0.0023 x (T_mean + 17.8) x sqrt(T_max - T_min) x Ra
        """
        t_mean = (temp_max + temp_min) / 2
        t_range = max(temp_max - temp_min, 0.1)

        # Extraterrestrial radiation (simplified)
        ra = self._calculate_ra(latitude, day_of_year)

        et0 = 0.0023 * (t_mean + 17.8) * np.sqrt(t_range) * ra
        return max(et0, 0)

    def _calculate_ra(self, latitude: float, day_of_year: int) -> float:
        """Calculate extraterrestrial radiation (MJ/m²/day)."""
        lat_rad = np.radians(latitude)
        solar_dec = 0.409 * np.sin(2 * np.pi * day_of_year / 365 - 1.39)
        ws = np.arccos(-np.tan(lat_rad) * np.tan(solar_dec))
        dr = 1 + 0.033 * np.cos(2 * np.pi * day_of_year / 365)

        ra = (24 * 60 / np.pi) * 0.082 * dr * (
            ws * np.sin(lat_rad) * np.sin(solar_dec) +
            np.cos(lat_rad) * np.cos(solar_dec) * np.sin(ws)
        )
        return ra

    def calculate_etc(self, et0: float, growth_stage: str) -> float:
        """
        Crop evapotranspiration: ETc = ET0 x Kc
        """
        kc = self.CROP_COEFFICIENTS.get(growth_stage, 0.7)
        return et0 * kc

    def calculate_stress_factor(self, soil_moisture: float,
                                growth_stage: str) -> float:
        """
```

```
        Calculate stress factor Ks (0-1, where 1 = no stress).
        """
        taw = (self.FIELD_CAPACITY - self.WILTING_POINT) * self.ROOT_DEPTH * 1000
        raw = 0.5 * taw  # Readily available water

        current_water = (soil_moisture / 100) * self.ROOT_DEPTH * 1000
        depletion = (self.FIELD_CAPACITY * self.ROOT_DEPTH * 1000) - current_water

        if depletion <= raw:
            return 1.0  # No stress
        elif depletion >= taw:
            return 0.0  # Max stress
        else:
            return (taw - depletion) / (taw - raw)

    def predict_stress_index(self, soil_moisture: float, temperature: float,
                             growth_stage: str, days_since_rain: int) -> float:
        """
        Convert stress factor to 0-100 stress index.
        """
        ks = self.calculate_stress_factor(soil_moisture, growth_stage)

        # Base stress from water deficit
        base_stress = (1 - ks) * 70

        # Temperature stress (optimal 25-30degC for maize)
        if temperature > 35:
            temp_stress = (temperature - 35) * 3
        elif temperature < 15:
            temp_stress = (15 - temperature) * 2
        else:
            temp_stress = 0

        # Drought duration factor
        duration_stress = min(days_since_rain * 2, 20)

        # Growth stage sensitivity multiplier
        stage_multipliers = {
            'EMERGENCE': 0.8,
            'VEGETATIVE': 0.9,
            'TASSELING': 1.3,  # Most sensitive
            'GRAIN_FILL': 1.1,
            'MATURITY': 0.7
        }
        multiplier = stage_multipliers.get(growth_stage, 1.0)

        stress_index = (base_stress + temp_stress + duration_stress) * multiplier
        return min(max(stress_index, 0), 100)

    def run_simulation(self, initial_state: dict, scenario: dict,
                       weather_forecast: list, days: int = 7) -> list:
        """
        Simulate stress trajectory over time.
        """
        trajectory = []
        soil_moisture = initial_state['soil_moisture']
```

```python
        for day in range(days):
            weather = weather_forecast[day] if day < len(weather_forecast) else weather_forecast[-1]

            # Calculate water loss
            et0 = self.calculate_et0(
                weather['temp_max'],
                weather['temp_min'],
                initial_state.get('latitude', 10.0),
                datetime.now().timetuple().tm_yday + day
            )
            etc = self.calculate_etc(et0, initial_state['growth_stage'])

            # Apply scenario modifications
            rainfall = scenario.get('rainfall_mm', weather.get('precipitation', 0)) / days
            irrigation = scenario.get('irrigation_mm', 0) / days

            # Update soil moisture (simplified)
            soil_moisture += (rainfall + irrigation - etc * 0.5)
            soil_moisture = max(min(soil_moisture, 100), 5)

            # Calculate stress
            stress = self.predict_stress_index(
                soil_moisture,
                weather['temp_max'],
                initial_state['growth_stage'],
                initial_state.get('days_since_rain', 0) + day
            )

            trajectory.append({
                'day': day + 1,
                'stress': round(stress),
                'soil_moisture': round(soil_moisture, 1)
            })

        return trajectory
```

4.2 Random Forest Classifier

```python
# models/random_forest.py

import numpy as np
from sklearn.ensemble import RandomForestClassifier
import joblib
from pathlib import Path

class StressRandomForest:
    """
    Random Forest classifier for stress category prediction.
    """

    FEATURES = [
        'soil_moisture',
        'temperature',
        'humidity',
        'days_since_rain',
        'growth_stage_encoded',
        'cumulative_et',
```

```python
        'forecast_rainfall_7d'
    ]

    CATEGORIES = ['NONE', 'LOW', 'MODERATE', 'SEVERE', 'CRITICAL']

    def __init__(self, model_path: str = None):
        if model_path and Path(model_path).exists():
            self.model = joblib.load(model_path)
        else:
            self.model = self._create_default_model()

    def _create_default_model(self) -> RandomForestClassifier:
        """Create and train on synthetic data for hackathon."""
        model = RandomForestClassifier(
            n_estimators=100,
            max_depth=10,
            random_state=42
        )

        # Generate synthetic training data
        X, y = self._generate_synthetic_data(1000)
        model.fit(X, y)

        return model

    def _generate_synthetic_data(self, n_samples: int):
        """Generate realistic synthetic training data."""
        np.random.seed(42)

        X = np.zeros((n_samples, len(self.FEATURES)))
        y = np.zeros(n_samples, dtype=int)

        for i in range(n_samples):
            soil_moisture = np.random.uniform(10, 80)
            temperature = np.random.uniform(20, 42)
            humidity = np.random.uniform(20, 90)
            days_since_rain = np.random.randint(0, 21)
            growth_stage = np.random.randint(0, 5)
            cumulative_et = np.random.uniform(20, 100)
            forecast_rain = np.random.uniform(0, 50)

            X[i] = [soil_moisture, temperature, humidity, days_since_rain,
                    growth_stage, cumulative_et, forecast_rain]

            # Determine category based on features
            stress_score = (
                (80 - soil_moisture) * 0.4 +
                max(temperature - 30, 0) * 2 +
                days_since_rain * 1.5 -
                forecast_rain * 0.3
            )

            if stress_score < 15:
                y[i] = 0  # NONE
            elif stress_score < 30:
                y[i] = 1  # LOW
            elif stress_score < 50:
```

```python
                y[i] = 2  # MODERATE
            elif stress_score < 70:
                y[i] = 3  # SEVERE
            else:
                y[i] = 4  # CRITICAL

        return X, y

    def encode_growth_stage(self, stage: str) -> int:
        stages = ['EMERGENCE', 'VEGETATIVE', 'TASSELING', 'GRAIN_FILL', 'MATURITY']
        return stages.index(stage) if stage in stages else 1

    def predict(self, features: dict) -> dict:
        """Predict stress category with probability."""
        X = np.array([[
            features['soil_moisture'],
            features['temperature'],
            features['humidity'],
            features['days_since_rain'],
            self.encode_growth_stage(features['growth_stage']),
            features.get('cumulative_et', 50),
            features.get('forecast_rainfall_7d', 0)
        ]])

        proba = self.model.predict_proba(X)[0]
        category_idx = np.argmax(proba)

        # Convert to stress index (center of category range)
        stress_ranges = [(0, 20), (21, 40), (41, 60), (61, 80), (81, 100)]
        stress_range = stress_ranges[category_idx]
        stress_index = (stress_range[0] + stress_range[1]) / 2

        return {
            'stress_index': stress_index,
            'category': self.CATEGORIES[category_idx],
            'confidence': float(proba[category_idx]),
            'probabilities': {cat: float(p) for cat, p in zip(self.CATEGORIES, proba)}
        }

    def save(self, path: str):
        joblib.dump(self.model, path)
```

4.3 LSTM Time Series Model

```python
# models/lstm.py

import numpy as np
from typing import List, Dict

try:
    import tensorflow as tf
    from tensorflow.keras.models import Sequential, load_model
    from tensorflow.keras.layers import LSTM, Dense, Dropout
    HAS_TF = True
except ImportError:
    HAS_TF = False
```

```python
class StressLSTM:
    """
    LSTM model for time-series stress forecasting.
    """

    SEQUENCE_LENGTH = 14  # 14 days of historical data
    FORECAST_DAYS = 7     # Predict 7 days ahead

    def __init__(self, model_path: str = None):
        if not HAS_TF:
            self.model = None
            return

        if model_path:
            try:
                self.model = load_model(model_path)
            except:
                self.model = self._create_model()
        else:
            self.model = self._create_model()

    def _create_model(self):
        """Create LSTM architecture."""
        if not HAS_TF:
            return None

        model = Sequential([
            LSTM(64, input_shape=(self.SEQUENCE_LENGTH, 4), return_sequences=True),
            Dropout(0.2),
            LSTM(32, return_sequences=False),
            Dropout(0.2),
            Dense(16, activation='relu'),
            Dense(self.FORECAST_DAYS, activation='linear')
        ])

        model.compile(optimizer='adam', loss='mse', metrics=['mae'])

        # Quick train on synthetic data
        X, y = self._generate_sequences(500)
        model.fit(X, y, epochs=10, batch_size=32, verbose=0)

        return model

    def _generate_sequences(self, n_samples: int):
        """Generate synthetic time-series data."""
        np.random.seed(42)

        X = np.zeros((n_samples, self.SEQUENCE_LENGTH, 4))
        y = np.zeros((n_samples, self.FORECAST_DAYS))

        for i in range(n_samples):
            # Generate a stress trajectory
            base_stress = np.random.uniform(20, 60)
            trend = np.random.uniform(-0.5, 1.5)  # Increasing or decreasing
            noise = np.random.normal(0, 3, self.SEQUENCE_LENGTH + self.FORECAST_DAYS)

            full_trajectory = base_stress + trend * np.arange(self.SEQUENCE_LENGTH + self.FORECAST_DAYS) +
```

```python
            full_trajectory = np.clip(full_trajectory, 0, 100)

            # Features: stress, soil_moisture (inverse), temperature, days
            X[i, :, 0] = full_trajectory[:self.SEQUENCE_LENGTH]  # stress
            X[i, :, 1] = 80 - full_trajectory[:self.SEQUENCE_LENGTH] * 0.5  # soil moisture
            X[i, :, 2] = 25 + full_trajectory[:self.SEQUENCE_LENGTH] * 0.1  # temperature
            X[i, :, 3] = np.arange(self.SEQUENCE_LENGTH)  # time

            y[i] = full_trajectory[self.SEQUENCE_LENGTH:]

        # Normalize
        X = X / 100.0
        y = y / 100.0

        return X, y

    def predict(self, historical_data: List[Dict]) -> Dict:
        """
        Predict future stress from historical data.

        historical_data: List of {timestamp, soil_moisture, temperature, stress_index}
        """
        if not HAS_TF or self.model is None:
            return self._fallback_predict(historical_data)

        # Prepare sequence
        sequence = np.zeros((1, self.SEQUENCE_LENGTH, 4))

        for i, data in enumerate(historical_data[-self.SEQUENCE_LENGTH:]):
            idx = i + max(0, self.SEQUENCE_LENGTH - len(historical_data))
            sequence[0, idx, 0] = data.get('stress_index', 50) / 100.0
            sequence[0, idx, 1] = data.get('soil_moisture', 50) / 100.0
            sequence[0, idx, 2] = data.get('temperature', 30) / 100.0
            sequence[0, idx, 3] = i / self.SEQUENCE_LENGTH

        # Predict
        forecast = self.model.predict(sequence, verbose=0)[0] * 100
        forecast = np.clip(forecast, 0, 100)

        return {
            'forecast': [{'day': i+1, 'stress': round(float(s))} for i, s in enumerate(forecast)],
            'average_stress': float(np.mean(forecast)),
            'trend': 'increasing' if forecast[-1] > forecast[0] else 'decreasing'
        }

    def _fallback_predict(self, historical_data: List[Dict]) -> Dict:
        """Simple fallback when TensorFlow not available."""
        if not historical_data:
            return {'forecast': [{'day': i+1, 'stress': 50} for i in range(7)],
                    'average_stress': 50, 'trend': 'stable'}

        recent = [d.get('stress_index', 50) for d in historical_data[-7:]]
        avg = np.mean(recent)
        trend = (recent[-1] - recent[0]) / len(recent) if len(recent) > 1 else 0

        forecast = []
        for i in range(self.FORECAST_DAYS):
```

```python
            stress = avg + trend * (i + 1)
            stress = max(0, min(100, stress))
            forecast.append({'day': i + 1, 'stress': round(stress)})

        return {
            'forecast': forecast,
            'average_stress': float(np.mean([f['stress'] for f in forecast])),
            'trend': 'increasing' if trend > 0.5 else 'decreasing' if trend < -0.5 else 'stable'
        }

    def save(self, path: str):
        if self.model:
            self.model.save(path)
```

4.4 Ensemble Combiner

```python
# models/ensemble.py

from typing import Dict, List
from .water_balance import WaterBalanceModel
from .random_forest import StressRandomForest
from .lstm import StressLSTM

class EnsemblePredictor:
    """
    Combines physics, ML, and deep learning models for robust predictions.
    """

    WEIGHTS = {
        'physics': 0.40,
        'random_forest': 0.35,
        'lstm': 0.25
    }

    def __init__(self):
        self.physics_model = WaterBalanceModel()
        self.rf_model = StressRandomForest()
        self.lstm_model = StressLSTM()

    def predict(self, request: Dict) -> Dict:
        """
        Generate ensemble prediction.
        """
        current = request['current_data']
        farm = request['farm_info']
        weather = request.get('weather_forecast', [])
        history = request.get('historical_data', [])

        # Physics model prediction
        physics_stress = self.physics_model.predict_stress_index(
            soil_moisture=current['soil_moisture'],
            temperature=current['temperature'],
            growth_stage=farm['growth_stage'],
            days_since_rain=current['days_since_rain']
        )

        # Random Forest prediction
```

```python
rf_features = {
    'soil_moisture': current['soil_moisture'],
    'temperature': current['temperature'],
    'humidity': current['humidity'],
    'days_since_rain': current['days_since_rain'],
    'growth_stage': farm['growth_stage'],
    'cumulative_et': current.get('cumulative_et', 50),
    'forecast_rainfall_7d': sum(w.get('precipitation', 0) for w in weather[:7])
}
rf_result = self.rf_model.predict(rf_features)

# LSTM prediction
lstm_result = self.lstm_model.predict(history)
lstm_stress = lstm_result['average_stress']

# Weighted ensemble
ensemble_stress = (
    self.WEIGHTS['physics'] * physics_stress +
    self.WEIGHTS['random_forest'] * rf_result['stress_index'] +
    self.WEIGHTS['lstm'] * lstm_stress
)

# Calculate confidence from model agreement
stresses = [physics_stress, rf_result['stress_index'], lstm_stress]
std_dev = np.std(stresses)
confidence = max(0.5, 1 - (std_dev / 50))  # Higher agreement = higher confidence

# Determine risk category
risk_category = self._get_risk_category(ensemble_stress)

# Calculate days to critical
days_to_critical = self._estimate_days_to_critical(
    ensemble_stress,
    lstm_result['forecast'],
    weather
)

# Generate recommendation
recommendation = self._generate_recommendation(
    ensemble_stress,
    risk_category,
    days_to_critical,
    current,
    farm
)

# Build forecast
physics_forecast = self.physics_model.run_simulation(
    {**current, **farm},
    {'rainfall_mm': sum(w.get('precipitation', 0) for w in weather)},
    weather
)

# Blend forecasts
blended_forecast = []
for i in range(7):
    phys = physics_forecast[i]['stress'] if i < len(physics_forecast) else ensemble_stress
```

```
            lstm = lstm_result['forecast'][i]['stress'] if i < len(lstm_result['forecast']) else ensemble_s
            blended = self.WEIGHTS['physics'] * phys + (1 - self.WEIGHTS['physics']) * lstm
            blended_forecast.append({
                'day': i + 1,
                'stress': round(blended),
                'category': self._get_risk_category(blended)
            })

        return {
            'stress_index': round(ensemble_stress),
            'risk_category': risk_category,
            'confidence': round(confidence, 2),
            'days_to_critical': days_to_critical,
            'recommendation': recommendation,
            'forecast': blended_forecast,
            'model_contributions': {
                'physics': round(physics_stress),
                'random_forest': round(rf_result['stress_index']),
                'lstm': round(lstm_stress)
            }
        }

    def simulate(self, request: Dict) -> Dict:
        """
        Run what-if simulation.
        """
        current = request['current_state']
        scenario = request['scenario']
        params = request['parameters']
        weather = request.get('weather_forecast', [])

        # Baseline (no changes)
        baseline_trajectory = self.physics_model.run_simulation(
            current,
            {'rainfall_mm': sum(w.get('precipitation', 0) for w in weather)},
            weather,
            days=params.get('duration_days', 7)
        )

        # Simulated scenario
        scenario_params = self._build_scenario_params(scenario, params)
        simulated_trajectory = self.physics_model.run_simulation(
            current,
            scenario_params,
            weather,
            days=params.get('duration_days', 7)
        )

        baseline_final = baseline_trajectory[-1]['stress']
        simulated_final = simulated_trajectory[-1]['stress']

        # Estimate yield impact
        yield_impact = self._estimate_yield_impact(simulated_final, current['growth_stage'])
        baseline_yield = self._estimate_yield_impact(baseline_final, current['growth_stage'])

        return {
            'baseline': {
```

```
                'stress_index': baseline_final,
                'yield_impact': baseline_yield,
                'trajectory': [t['stress'] for t in baseline_trajectory]
            },
            'simulated': {
                'stress_index': simulated_final,
                'yield_impact': yield_impact,
                'trajectory': [t['stress'] for t in simulated_trajectory]
            },
            'recommendation': self._generate_simulation_recommendation(
                baseline_final, simulated_final, scenario, params
            ),
            'risk_window': {
                'critical_day': next(
                    (i+1 for i, t in enumerate(simulated_trajectory) if t['stress'] >= 80),
                    None
                ),
                'action_deadline': max(1, next(
                    (i for i, t in enumerate(simulated_trajectory) if t['stress'] >= 80),
                    len(simulated_trajectory)
                ) - 1)
            }
        }

    def _get_risk_category(self, stress: float) -> str:
        if stress <= 20: return 'NONE'
        if stress <= 40: return 'LOW'
        if stress <= 60: return 'MODERATE'
        if stress <= 80: return 'SEVERE'
        return 'CRITICAL'

    def _estimate_days_to_critical(self, current_stress: float,
                                    forecast: List[Dict], weather: List[Dict]) -> int:
        for i, f in enumerate(forecast):
            if f['stress'] >= 80:
                return i + 1
        return None  # Not expected to reach critical

    def _generate_recommendation(self, stress: float, category: str,
                                  days_critical: int, current: Dict, farm: Dict) -> str:
        if category == 'NONE':
            return "Crop is healthy. Continue monitoring."
        elif category == 'LOW':
            return "Mild stress detected. Monitor soil moisture closely."
        elif category == 'MODERATE':
            return f"Moderate stress. Consider irrigating 15-20mm within {days_critical or 5} days."
        elif category == 'SEVERE':
            irrigation = 25 if farm['growth_stage'] == 'TASSELING' else 20
            return f"Severe stress! Irrigate {irrigation}mm within {days_critical or 3} days to prevent yie
        else:
            return "Critical stress! Irrigate immediately (30mm+) to minimize crop damage."

    def _build_scenario_params(self, scenario: str, params: Dict) -> Dict:
        if scenario == 'DRY_WEEK':
            return {'rainfall_mm': 0, 'duration_days': params.get('duration_days', 14)}
        elif scenario == 'IRRIGATION_TEST':
            return {'irrigation_mm': params.get('irrigation_mm', 20)}
```

```python
        elif scenario == 'DELAYED_PLANTING':
            return {'delay_days': params.get('delay_days', 14)}
        return params

    def _estimate_yield_impact(self, stress: float, growth_stage: str) -> int:
        """Estimate % yield loss based on stress and growth stage."""
        stage_sensitivity = {
            'EMERGENCE': 0.5,
            'VEGETATIVE': 0.7,
            'TASSELING': 1.5,  # Most sensitive
            'GRAIN_FILL': 1.2,
            'MATURITY': 0.4
        }

        if stress <= 40:
            return 0

        multiplier = stage_sensitivity.get(growth_stage, 1.0)
        base_impact = (stress - 40) * 0.6 * multiplier

        return -min(round(base_impact), 80)

    def _generate_simulation_recommendation(self, baseline: float, simulated: float,
                                             scenario: str, params: Dict) -> str:
        diff = simulated - baseline

        if diff <= 5:
            return "Minimal impact expected from this scenario."

        if scenario == 'DRY_WEEK':
            days = params.get('duration_days', 14)
            return f"A {days}-day dry period would increase stress by {round(diff)} points. Irrigate 25mm b
        elif scenario == 'IRRIGATION_TEST':
            amount = params.get('irrigation_mm', 20)
            return f"Adding {amount}mm irrigation would reduce stress by {round(-diff)} points."
        elif scenario == 'DELAYED_PLANTING':
            delay = params.get('delay_days', 14)
            return f"Delaying planting by {delay} days would increase peak stress by {round(diff)} points."

        return f"This scenario increases stress by {round(diff)} points. Take preventive action."


import numpy as np  # Add at top of file if not present
```

---

5. FastAPI Application

```python
# app/main.py

from fastapi import FastAPI, HTTPException
from fastapi.middleware.cors import CORSMiddleware
from pydantic import BaseModel
from typing import List, Dict, Optional
import uvicorn

from models.ensemble import EnsemblePredictor
```

```python
app = FastAPI(
    title="Demeter ML Service",
    description="AI-powered crop stress prediction and simulation",
    version="1.0.0"
)

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
    allow_methods=["*"],
    allow_headers=["*"],
)

# Initialize ensemble
predictor = EnsemblePredictor()


class CurrentData(BaseModel):
    soil_moisture: float
    temperature: float
    humidity: float
    days_since_rain: int

class FarmInfo(BaseModel):
    planting_date: str
    growth_stage: str
    latitude: float = 10.5
    longitude: float = 7.4

class WeatherDay(BaseModel):
    date: str
    temp_max: float
    temp_min: float
    precipitation: float = 0

class HistoricalData(BaseModel):
    timestamp: str
    soil_moisture: float
    temperature: float
    stress_index: Optional[float] = None

class PredictionRequest(BaseModel):
    farm_id: str
    current_data: CurrentData
    farm_info: FarmInfo
    weather_forecast: List[WeatherDay] = []
    historical_data: List[HistoricalData] = []

class SimulationRequest(BaseModel):
    farm_id: str
    current_state: Dict
    scenario: str
    parameters: Dict
    weather_forecast: List[WeatherDay] = []


@app.get("/health")
```

```python
def health_check():
    return {"status": "healthy", "service": "demeter-ml"}


@app.post("/predict")
def predict(request: PredictionRequest):
    try:
        result = predictor.predict(request.dict())
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


@app.post("/simulate")
def simulate(request: SimulationRequest):
    try:
        result = predictor.simulate(request.dict())
        return result
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))


if __name__ == "__main__":
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

---

6. Project Structure

```
ml-service/
||| app/
|   ||| __init__.py
|   ||| main.py                # FastAPI application
|   ||| config.py              # Configuration
||| models/
|   ||| __init__.py
|   ||| water_balance.py       # Physics model
|   ||| random_forest.py       # RF classifier
|   ||| lstm.py                # Time series model
|   ||| ensemble.py            # Ensemble combiner
||| synthetic/
|   ||| __init__.py
|   ||| data_generator.py      # Synthetic data for training
||| tests/
|   ||| test_water_balance.py
|   ||| test_ensemble.py
|   ||| test_api.py
||| trained_models/            # Saved model weights
|   ||| rf_model.pkl
|   ||| lstm_model.h5
||| requirements.txt
||| Dockerfile
||| README.md
```

---

7. Dependencies

```
# requirements.txt
fastapi==0.109.0
uvicorn==0.27.0
pydantic==2.5.0
numpy==1.26.0
pandas==2.1.0
scikit-learn==1.4.0
tensorflow==2.15.0  # Optional, fallback exists
joblib==1.3.0
python-multipart==0.0.6
httpx==0.26.0
```

---

8. Deliverables Checklist

- ☐ Water balance model with Hargreaves ET0
- ☐ Random Forest stress classifier
- ☐ LSTM time series forecaster (with fallback)
- ☐ Ensemble combiner with weighted averaging
- ☐ FastAPI endpoints (/predict, /simulate, /health)
- ☐ Synthetic data generator for training
- ☐ Confidence scoring from model agreement
- ☐ Recommendation generation
- ☐ Yield impact estimation
- ☐ Unit tests for models
- ☐ Docker setup

---

9. Testing the API

```bash
# Start server
uvicorn app.main:app --reload --port 8000

# Test prediction
curl -X POST http://localhost:8000/predict \
  -H "Content-Type: application/json" \
  -d '{
    "farm_id": "test-farm",
    "current_data": {
      "soil_moisture": 32,
      "temperature": 34,
      "humidity": 45,
      "days_since_rain": 5
    },
    "farm_info": {
      "planting_date": "2026-02-01",
      "growth_stage": "VEGETATIVE",
      "latitude": 10.51,
      "longitude": 7.42
    }
  }'
```

Coordinate with: Backend (API integration), Hardware (sensor data format)