PRODUCT REQUIREMENTS DOCUMENT (PRD)

Digital Bird Stress Twin: AI-Powered Disaster Early Warning System

---

Document Version: 1.0
Last Updated: December 17, 2025
Author: Anmol Negi
Project Duration: 21 Days
Project Type: Production-Grade MLOps System
Confidentiality: Public (Portfolio Project)

DOCUMENT CONTROL

Version	Date	Author	Changes
1.0	Dec 17, 2025	Anmol Negi	Initial PRD Creation

Reviewers:

- Technical Lead: [To be assigned]
- ML Architect: [To be assigned]
- DevOps Engineer: [To be assigned]

Approvers:

- Project Owner: Anmol Negi
- Technical Reviewer: [Pending]

TABLE OF CONTENTS

---

1. EXECUTIVE SUMMARY

1.1 Project Vision

The Digital Bird Stress Twin is an end-to-end production-grade AI system that predicts natural disasters 24-72 hours in advance by monitoring and analyzing avian stress behavior through bioacoustic patterns and environmental data.

1.2 Problem Statement

Current Challenge:

- Traditional disaster early warning systems rely solely on physical sensors and meteorological models
- Limited lead time (often <12 hours) for community preparedness
- High false alarm rates causing warning fatigue
- Expensive infrastructure requirements

Opportunity:

- Scientific research shows birds exhibit measurable stress behaviors 24-72 hours before natural disasters
- Bioacoustic analysis combined with environmental data provides complementary early warning signals
- Modern ML/DL techniques enable real-time pattern detection at scale

Innovation:

- First-of-its-kind "Digital Twin" approach to avian stress modeling
- Combines time-series deep learning, generative AI, and production MLOps
- Continuous learning system that improves over time
- Scalable, cost-effective early warning infrastructure

1.3 Project Objectives

Primary Objectives:

1. Build a production-ready LSTM-based stress prediction model achieving >85% accuracy
2. Implement complete MLOps pipeline with automated retraining and monitoring
3. Deploy scalable API serving real-time predictions with <500ms latency
4. Create generative acoustic model for stress behavior simulation

Secondary Objectives:

1. Demonstrate mastery of production ML engineering skills
2. Create portfolio-grade project for 10-15 LPA job applications
3. Contribute novel approach to disaster prediction research
4. Build reusable MLOps framework for future projects

1.4 Success Metrics

Technical KPIs:

- Model Accuracy: ≥85% on test set
- Precision (High/Critical Risk): ≥80% (minimize false alarms)
- Recall (High/Critical Risk): ≥90% (detect most disasters)
- Prediction Lead Time: 24-72 hours before event
- API Latency: <500ms (p95)
- System Uptime: ≥99%
- Model Drift Detection: Automated alerts within 24 hours

Business KPIs:

- Complete working demo ready in 21 days
- GitHub repository with >100 commits

- Technical blog post with >1000 views
- 3+ job interviews secured using this project
- Portfolio presentation ready for technical rounds

1.5 Out of Scope (v1.0)

The following features are explicitly excluded from initial release:

- Real-time streaming data processing (batch predictions only)
- Mobile application development
- Multi-language support (English only)
- Integration with government disaster management systems
- Historical disaster prediction backfilling
- Multi-species ensemble models (focus on crows initially)

---

2. PROJECT OVERVIEW

2.1 Background & Context

Scientific Foundation: Research has documented anomalous animal behavior before natural disasters:

- Birds show increased agitation and altered vocalization patterns 24-72h before earthquakes
- Changes in atmospheric pressure affect bird stress levels
- Electromagnetic field anomalies (before earthquakes) affect avian navigation and behavior
- Published studies: Wikelski et al. (2020), Grant & Halliday (2010)

Technical Context:

- Modern bioacoustics allows large-scale bird call analysis
- Deep learning enables pattern recognition in complex temporal data
- MLOps practices ensure production-quality ML systems
- Cloud infrastructure enables scalable, cost-effective deployment

Market Context:

- Growing demand for ML Engineers with production deployment experience
- MLOps skills are in top 5 most sought-after tech skills (LinkedIn 2024)
- Real-world portfolio projects significantly increase job offer rates

- Interdisciplinary AI applications (ecology + ML) are increasingly valued

2.2 Stakeholders

Primary Stakeholders:

1. **Anmol Negi (Project Owner)**
   - Responsibility: Overall project execution, technical decisions, documentation
   - Success Criteria: Completed project demonstrating production ML skills
2. **Potential Employers (Target Companies)**
   - Needs: Evidence of production ML engineering capabilities
   - Evaluation Criteria: Code quality, system design, MLOps practices

Secondary Stakeholders: 3. **Technical Community**

- Medium/Dev.to readers seeking MLOps tutorials
- GitHub users looking for production ML examples

4. **Research Community**
   - Academics interested in bioacoustics and disaster prediction
   - Potential collaboration opportunities

2.3 Assumptions & Constraints

Assumptions:

1. eBird and Xeno-Canto APIs will remain accessible and free
2. GCP free tier ($300 credit) is sufficient for development and demo
3. Historical disaster data can be compiled from public sources
4. Bird stress patterns are consistent across similar species
5. Internet connectivity is available for API deployment

Constraints:

1. **Time:** 21-day development timeline
2. **Budget:** $0 out-of-pocket (free tier services only)
3. **Compute:** Single GPU (personal laptop or Colab)
4. **Data:** Limited to publicly available datasets

5. **Expertise:** Solo developer (no team support)

2.4 Dependencies

External Dependencies:

- eBird API (bird observation data)
- Xeno-Canto API (bird call audio)
- OpenWeather API / IMD (weather data)
- USGS Earthquake Database (disaster events)
- GCP Cloud Run (deployment platform)
- GitHub (version control and CI/CD)

Technical Dependencies:

- Python 3.9+
- PyTorch 2.0+
- Docker
- PostgreSQL / MongoDB
- Various Python libraries (see Technical Stack)

---

3. BUSINESS REQUIREMENTS

3.1 Functional Requirements

FR-1: Data Collection System

Priority: CRITICAL
Description: System must automatically collect bird observation, audio, and environmental data daily

Requirements:

- FR-1.1: Fetch bird observations from eBird API for specified regions
- FR-1.2: Download bird call recordings from Xeno-Canto (minimum 50 calls/day)
- FR-1.3: Collect hourly weather data (pressure, temperature, humidity, wind)
- FR-1.4: Retrieve historical disaster events for model training
- FR-1.5: Validate data quality and completeness
- FR-1.6: Store raw data with version control (DVC)

- FR-1.7: Handle API rate limits and failures gracefully
- FR-1.8: Log all data collection activities

Acceptance Criteria:

- Successfully fetches data for 5+ locations daily
- 95%+ data collection success rate
- All data timestamped and georeferenced
- Failed requests retry up to 3 times

FR-2: Feature Engineering Pipeline

Priority: CRITICAL
Description: Extract meaningful features from raw audio and environmental data

Requirements:

- FR-2.1: Extract MFCC features from bird call audio (13 coefficients)
- FR-2.2: Calculate spectral entropy, ZCR, spectral centroid/rolloff
- FR-2.3: Engineer temporal features (lags, rolling statistics)
- FR-2.4: Create pressure gradient features (1h, 6h, 24h drops)
- FR-2.5: Generate interaction features (weather × time)
- FR-2.6: Normalize/standardize all features
- FR-2.7: Handle missing values appropriately
- FR-2.8: Create feature documentation with statistical summaries

Acceptance Criteria:

- Feature extraction completes for 100+ audio files in <10 minutes
- Zero NaN values in final feature set
- Feature distributions documented and validated
- Reproducible feature engineering with versioned pipeline

FR-3: Stress Prediction Model

Priority: CRITICAL
Description: LSTM-based model to predict bird stress levels from temporal data

Requirements:

- FR-3.1: Train LSTM model on 168-hour sequences

- FR-3.2: Input: 92 features (audio + weather + temporal)
- FR-3.3: Output: Stress score [0,1] + uncertainty estimate
- FR-3.4: Achieve ≥85% accuracy on test set
- FR-3.5: Achieve ≥90% recall for high-stress events
- FR-3.6: Provide attention weights for interpretability
- FR-3.7: Support batch inference
- FR-3.8: Model inference time <500ms per prediction

Acceptance Criteria:

- Model passes all accuracy thresholds on hold-out test set
- Confusion matrix shows balanced performance across stress levels
- Model generalizes to unseen locations
- Uncertainty estimates are well-calibrated

FR-4: Risk Classification System

Priority: HIGH
Description: Classify disaster risk level and type from stress predictions

Requirements:

- FR-4.1: Map stress scores to risk levels (LOW/MEDIUM/HIGH/CRITICAL)
- FR-4.2: Predict disaster type (EARTHQUAKE/FLOOD/STORM/CYCLONE/NONE)
- FR-4.3: Provide confidence scores for predictions
- FR-4.4: Consider environmental context in classification
- FR-4.5: Generate human-readable risk explanations

Acceptance Criteria:

- Risk levels align with actual disaster severity
- Disaster type prediction accuracy ≥70%
- Clear decision boundaries documented
- Minimal false CRITICAL alarms (<5%)

FR-5: Generative Acoustic Model (VAE)

Priority: MEDIUM
Description: Generate synthetic bird calls for stress simulation

Requirements:

- FR-5.1: Train VAE on bird call MFCC features
- FR-5.2: Condition generation on stress level
- FR-5.3: Generate realistic acoustic features
- FR-5.4: Visualize latent space representation
- FR-5.5: Enable interpolation between stress levels

Acceptance Criteria:

- Generated features are statistically similar to real data
- Smooth transitions in latent space between stress levels
- Model can generate 100 samples in <10 seconds

FR-6: Prediction API

Priority: CRITICAL
Description: RESTful API for serving model predictions

Requirements:

- FR-6.1: POST /predict/stress endpoint (single prediction)
- FR-6.2: POST /predict/batch endpoint (multiple predictions)
- FR-6.3: GET /health endpoint (system health)
- FR-6.4: GET /metrics endpoint (model performance)
- FR-6.5: GET /model/version endpoint (current model info)
- FR-6.6: Input validation with Pydantic models
- FR-6.7: Error handling with meaningful messages
- FR-6.8: Request logging for monitoring

Acceptance Criteria:

- All endpoints return within 500ms (p95)
- API handles 100 concurrent requests
- Returns HTTP status codes correctly
- Input validation catches malformed requests
- Swagger documentation auto-generated

FR-7: Model Training Pipeline

Priority: CRITICAL
Description: Automated pipeline for model training and retraining

Requirements:

- FR-7.1: Orchestrated with Airflow (DAG-based)
- FR-7.2: Data preparation and splitting (temporal split)
- FR-7.3: Hyperparameter configuration
- FR-7.4: Model training with progress logging
- FR-7.5: Model evaluation on validation set
- FR-7.6: Model registration with MLflow
- FR-7.7: Conditional model promotion (if improved)
- FR-7.8: Automated retraining weekly

Acceptance Criteria:

- Training pipeline runs end-to-end without manual intervention
- All experiments logged to MLflow
- Models versioned and tagged appropriately
- Training completes in <4 hours

FR-8: Model Monitoring System

Priority: HIGH
Description: Continuous monitoring for data/prediction drift

Requirements:

- FR-8.1: Monitor feature distribution drift (Evidently AI)
- FR-8.2: Monitor prediction distribution drift
- FR-8.3: Track model performance metrics (accuracy, latency)
- FR-8.4: Alert on drift threshold breach
- FR-8.5: Generate weekly monitoring reports
- FR-8.6: Dashboard for real-time metrics (Grafana)
- FR-8.7: Log prediction history for analysis

Acceptance Criteria:

- Drift detection alerts within 24 hours
- Monitoring dashboard accessible 24/7
- Historical metrics retained for 90 days
- False alert rate <10%

3.2 Non-Functional Requirements

NFR-1: Performance

- **NFR-1.1:** API response time <500ms for 95% of requests
- **NFR-1.2:** Model inference time <200ms on CPU
- **NFR-1.3:** Feature extraction processes 100 audio files in <10 minutes
- **NFR-1.4:** Dashboard loads in <3 seconds
- **NFR-1.5:** Training pipeline completes in <4 hours

NFR-2: Scalability

- **NFR-2.1:** API handles 100 concurrent requests
- **NFR-2.2:** Database supports 1M+ feature records
- **NFR-2.3:** Feature engineering pipeline processes 1000+ audio files/day
- **NFR-2.4:** System scales to 50+ monitoring locations

NFR-3: Reliability

- **NFR-3.1:** System uptime ≥99% (excluding maintenance)
- **NFR-3.2:** Automated retry for failed API calls (max 3 attempts)
- **NFR-3.3:** Graceful degradation if external APIs unavailable
- **NFR-3.4:** Data backup every 24 hours
- **NFR-3.5:** Model rollback capability if new version fails

NFR-4: Maintainability

- **NFR-4.1:** Modular codebase with clear separation of concerns
- **NFR-4.2:** All functions have docstrings (Google style)
- **NFR-4.3:** Code coverage ≥80%
- **NFR-4.4:** Comprehensive README with setup instructions
- **NFR-4.5:** Type hints on all function signatures

- **NFR-4.6:** Logging at appropriate levels (DEBUG, INFO, WARNING, ERROR)

NFR-5: Usability

- **NFR-5.1:** API documentation auto-generated (Swagger/OpenAPI)
- **NFR-5.2:** Clear error messages for API users
- **NFR-5.3:** Monitoring dashboard intuitive for non-technical users
- **NFR-5.4:** Model predictions include confidence intervals
- **NFR-5.5:** Risk levels color-coded (GREEN/YELLOW/ORANGE/RED)

NFR-6: Security

- **NFR-6.1:** API keys stored in environment variables (not code)
- **NFR-6.2:** No sensitive data in logs
- **NFR-6.3:** HTTPS for all API endpoints
- **NFR-6.4:** Input validation prevents injection attacks
- **NFR-6.5:** Docker containers run as non-root user

NFR-7: Portability

- **NFR-7.1:** Fully containerized with Docker
- **NFR-7.2:** Works on Linux, macOS, Windows (via Docker)
- **NFR-7.3:** Cloud-agnostic (can deploy to GCP, AWS, Azure)
- **NFR-7.4:** No hard-coded file paths

3.3 User Stories

US-1: As a hiring manager, I want to:

- View a live demo of the system making predictions
- See clean, documented code on GitHub
- Understand the technical architecture through diagrams
- Review model performance metrics
- **Acceptance:** Can assess candidate's ML engineering skills in 30 minutes

US-2: As Anmol (developer), I want to:

- Train models with different hyperparameters easily

- Compare experiment results in one place (MLflow)
- Deploy model updates with one command
- Debug production issues quickly through logs
- **Acceptance:** Can iterate on models efficiently

US-3: As a technical interviewer, I want to:

- Ask questions about specific design decisions
- Understand trade-offs made during development
- Verify understanding of underlying ML concepts
- See evidence of production ML best practices
- **Acceptance:** Can conduct 45-min deep-dive technical interview

---

4. TECHNICAL REQUIREMENTS

4.1 Technology Stack

4.1.1 Programming Languages

- **Python 3.9+**: Primary development language
  - Reason: Best ML/AI ecosystem support
  - Libraries: PyTorch, scikit-learn, pandas, numpy

4.1.2 Machine Learning Frameworks

- **PyTorch 2.0+**: Deep learning models
  - Reason: Dynamic computation graphs, production-ready
- **scikit-learn 1.3+**: Feature preprocessing, metrics
- **librosa 0.10+**: Audio signal processing
- **Hugging Face Transformers**: (Optional) If using pre-trained models

4.1.3 MLOps Tools

- **MLflow 2.8+**: Experiment tracking, model registry
  - Reason: Industry standard, comprehensive features
- **DVC 3.0+**: Data version control

    - Reason: Git-like workflow for datasets
- **Apache Airflow 2.7+**: Workflow orchestration
    - Reason: Robust DAG-based pipeline management
- **Evidently AI 0.4+**: ML monitoring
    - Reason: Purpose-built for ML model monitoring

4.1.4 API & Web Frameworks

- **FastAPI 0.104+**: REST API framework
    - Reason: High performance, auto-documentation, type safety
- **Pydantic 2.5+**: Data validation
- **Uvicorn**: ASGI server

4.1.5 Data Storage

- **PostgreSQL 15+**: Structured data (metadata, predictions)
    - Reason: ACID compliance, excellent time-series support
- **MongoDB 7.0+**: (Optional) Unstructured data storage
- **AWS S3 / GCS**: Audio file storage, model artifacts
    - Reason: Cost-effective, scalable object storage

4.1.6 Containerization & Orchestration

- **Docker 24+**: Application containerization
- **Docker Compose 2.0+**: Local multi-container orchestration

4.1.7 Cloud Platform

- **Google Cloud Platform (GCP)**:
    - Cloud Run: Serverless container deployment
    - Cloud Storage: Object storage
    - Container Registry: Docker image storage
    - Reason: $300 free credit, serverless simplicity

4.1.8 CI/CD

- **GitHub Actions**: Automated testing and deployment

  - Reason: Free for public repos, integrated with GitHub

4.1.9 Monitoring & Observability

- **Prometheus**: Metrics collection
- **Grafana**: Metrics visualization
- **Python logging**: Application logs

4.1.10 Development Tools

- **Git**: Version control
- **GitHub**: Code hosting, collaboration
- **VS Code / PyCharm**: IDE
- **Jupyter Notebooks**: Exploratory analysis
- **pytest**: Unit testing
- **Black**: Code formatting
- **Pylint**: Code linting

4.2 Development Environment Setup

4.2.1 Hardware Requirements

Minimum:

- CPU: 4 cores
- RAM: 16 GB
- Storage: 50 GB free space
- GPU: Optional (can use Google Colab)

Recommended:

- CPU: 8 cores
- RAM: 32 GB
- Storage: 100 GB SSD
- GPU: NVIDIA with 8+ GB VRAM (e.g., RTX 3070)

4.2.2 Software Requirements

- Operating System: Ubuntu 22.04 / macOS 12+ / Windows 11 (WSL2)

- Docker Desktop
- Python 3.9+ with pip
- Git
- PostgreSQL client tools
- gcloud CLI (for GCP deployment)

4.2.3 Python Environment

```bash
# Create virtual environment
python3.9 -m venv venv
source venv/bin/activate  # Linux/Mac
# or
venv\Scripts\activate  # Windows

# Install dependencies
pip install -r requirements.txt
```

requirements.txt (minimum):

```
# ML/DL
torch==2.1.0
torchvision==0.16.0
scikit-learn==1.3.2
numpy==1.24.3
pandas==2.1.3

# Audio processing
librosa==0.10.1
soundfile==0.12.1

# MLOps
mlflow==2.8.1
dvc==3.30.1
evidently==0.4.11

# API
fastapi==0.104.1
uvicorn==0.24.0
pydantic==2.5.0
python-multipart==0.0.6

# Data processing
scipy==1.11.4
requests==2.31.0
beautifulsoup4==4.12.2

# Visualization
matplotlib==3.8.2
seaborn==0.13.0
plotly==5.18.0

# Database
psycopg2-binary==2.9.9
sqlalchemy==2.0.23

# Workflow
apache-airflow==2.7.3

# Testing
pytest==7.4.3
pytest-cov==4.1.0

# Utilities
```

```
python-dotenv==1.0.0
tqdm==4.66.1
```

4.3 External API Requirements

4.3.1 eBird API

- **Purpose:** Bird observation data
- **Endpoint:** https://api.ebird.org/v2/
- **Authentication:** API key (free, request at ebird.org)
- **Rate Limits:** 100 requests/day (free tier)
- **Required Data:**
  - Recent observations by region
  - Species occurrence data
  - Observation counts

4.3.2 Xeno-Canto API

- **Purpose:** Bird call audio recordings
- **Endpoint:** https://xeno-canto.org/api/2/recordings
- **Authentication:** None required
- **Rate Limits:** Reasonable use (no official limit)
- **Required Data:**
  - Audio file URLs
  - Recording metadata (location, date, species)
  - Audio quality ratings

4.3.3 OpenWeather API

- **Purpose:** Weather data
- **Endpoint:** https://api.openweathermap.org/data/3.0/
- **Authentication:** API key (free tier: 1000 calls/day)
- **Rate Limits:** 60 calls/minute
- **Required Data:**
  - Current weather
  - Historical weather

- Hourly forecasts (optional)

Alternative: India Meteorological Department (IMD) API

- More accurate for India-specific data
- May require special access

4.3.4 USGS Earthquake API

- **Purpose:** Historical earthquake data
- **Endpoint:** https://earthquake.usgs.gov/fdsnws/event/1/
- **Authentication:** None required
- **Required Data:**
    - Earthquake events (magnitude, location, time)
    - Aftershock sequences

---

5. SYSTEM ARCHITECTURE

5.1 High-Level Architecture Diagram

```
┌─────────────────────────────────────────────────────────────┐
│                 EXTERNAL DATA LAYER                         │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌─────────────┐  ┌─────────────┐  ┌─────────────┐          │
│  │  eBird API  │  │ Xeno-Canto  │  │Weather APIs │          │
│  │ (Bird Obs.) │  │   (Audio)   │  │(IMD/OpenWx) │          │
│  └──────┬──────┘  └──────┬──────┘  └──────┬──────┘          │
│         │                │                │                 │
└─────────┼────────────────┼────────────────┼─────────────────┘
          │                │                │
          ↓                ↓                ↓
┌─────────────────────────────────────────────────────────────┐
│                 DATA INGESTION LAYER                        │
├─────────────────────────────────────────────────────────────┤
│                                                             │
│  ┌───────────────────────────────────────────────────────┐  │
│  │          Apache Airflow Orchestration                 │  │
│  │                                                       │  │
│  │  DAG 1: Daily Data Collection (2 AM)                  │  │
│  │  ├─ Fetch eBird observations                          │  │
│  │  ├── Download Xeno-Canto audio                    │ │
│  │  ├── Pull weather data                            │ │
│  │  └── Data quality validation                      │ │
│  │                                                   │ │
│  │ DAG 2: Weekly Model Training (Sunday 3 AM)        │ │
│  │  ├── Prepare training dataset                     │ │
│  │  ├── Train models                                 │ │
│  │  ├── Evaluate performance                         │ │
│  │  └── Register best model                          │ │
│  └────────────────────────────────────────────────────┘ │
│                  │                    │
│                  ↓                    │
│  ┌────────────────────────────────────────────────────┐ │
│  │         Raw Data Storage (DVC-versioned)          │ │
│  │ • Audio files → Cloud Storage (GCS)               │ │
│  │ • Observation data → PostgreSQL                   │ │
│  │ • Weather data → PostgreSQL                       │ │
│  └────────────────────────────────────────────────────┘ │
│                                  │
└──────────────────────────────────────┬────────────────────┘
                  ↓
┌──────────────────────────────────────────────────────────┐
│          PREDICTION SERVICE LAYER                        │
├──────────────────────────────────────────────────────────┤
│                                  │
│  ┌────────────────────────────────────────────────────┐ │
│  │             FastAPI Application                   │ │
│  │                                                   │ │
│  │ Endpoints:                                        │ │
│  │  ├── POST /predict/stress                         │ │
│  │  │    Input: audio_url, location, weather         │ │
│  │  │    Output: stress_score, risk_level, confidence │ │
│  │  │                                                │ │
│  │  ├── POST /predict/batch                          │ │
│  │  │    Input: list of prediction requests          │ │
│  │  │    Output: list of predictions                 │ │
│  │  │                                                │ │
│  │  ├── GET /health                                  │ │
│  │  │    Output: system status, model version        │ │
│  │  │                                                │ │
│  │  ├── GET /metrics                                 │ │
│  │  │    Output: request count, latency, accuracy    │ │
│  │  │                                                │ │
│  │  └── GET /model/version                           │ │
│  │       Output: current model metadata              │ │
│  │                                                   │ │
│  │ Components:                                       │ │
│  │  ├─ Input validation (Pydantic)             │ │
│  │  ├─ Model loader (from MLflow)                 │ │
│  │  ├─ Feature extraction                      │ │
│  │  ├─ Inference engine                        │ │
│  │  └─ Response formatting                       │ │
│  └─────────────────────────────────────────────────────────┘  │
│                                  │
└───────────────────────────────────┬───────────────────────────────┘
                ↓
┌───────────────────────────────────────────────────────────────────┐
│          MONITORING & OBSERVABILITY LAYER                │
├───────────────────────────────────────────────────────────────────┤
│                                  │
│  ┌─────────────────────────────────────────────────────────┐  │
│  │          Evidently AI - Model Monitoring                │ │
│  │  • Feature drift detection                        │ │
│  │  • Prediction drift detection                      │ │
│  │  • Data quality checks                           │ │
│  │  • Performance degradation alerts                    │ │
│  └─────────────────────────────────────────────────────────┘  │
│               │                        │
│  ┌─────────────────────────────────────────────────────────┐  │
│  │        Prometheus + Grafana - Metrics & Dashboards        │ │
│  │  • Request latency (p50, p95, p99)                  │ │
│  │  • Request rate (requests/sec)                    │ │
│  │  • Error rate                                 │ │
│  │  • Model inference time                          │ │
│  │  • Resource utilization (CPU, RAM)                  │ │
│  └─────────────────────────────────────────────────────────┘  │
│               │                        │
│  ┌─────────────────────────────────────────────────────────┐  │
│  │          Logging & Alerting System                   │ │
│  │  • Application logs (DEBUG, INFO, WARNING, ERROR)           │ │
│  │  • Prediction logs (input, output, timestamp)              │ │
│  │  • Error tracking and debugging                       │ │
│  │  • Slack/Email alerts on critical issues                 │ │
│  └─────────────────────────────────────────────────────────┘  │
│                                  │
└───────────────────────────────────┬───────────────────────────────┘
                ↓
┌───────────────────────────────────────────────────────────────────┐
│         DEPLOYMENT & INFRASTRUCTURE LAYER                 │
├───────────────────────────────────────────────────────────────────┤
│                                  │
│  ┌─────────────────────────────────────────────────────────┐  │
│  │             Docker Containers                   │ │
│  │  ┌──────────────┐  ┌──────────────┐  ┌──────────────┐      │ │
│  │  │  API      │  │  Airflow    │  │  Monitoring │      │  │
│  │  │  Service    │  │  Scheduler  │  │  Stack    │      │  │
│  │  └──────────────┘  └──────────────┘  └──────────────┘      │  │
│  └──────────────────────────────────────────────────────────┘  │
│                │                │
│  ┌──────────────────────────────────────────────────────────┐  │
│  │      Google Cloud Platform (GCP)         │  │
│  │  ┌──────────────────────────────────────────────────┐  │  │
│  │  │ Cloud Run (Serverless Container Platform)   │  │  │
│  │  │ • Auto-scaling (0 → N instances)       │  │  │
│  │  │ • Pay-per-request pricing          │  │  │
│  │  │ • HTTPS endpoint with SSL          │  │  │
│  │  │ • Environment variable management       │  │  │
│  │  └──────────────────────────────────────────────────┘  │  │
│  │                        │  │
│  │  ┌──────────────────────────────────────────────────┐  │  │
│  │  │ Cloud Storage (Object Storage)       │  │  │
│  │  │ • Audio file storage          │  │  │
│  │  │ • Model artifact storage        │  │  │
│  │  │ • Backup storage           │  │  │
│  │  └──────────────────────────────────────────────────┘  │  │
│  │                        │  │
│  │  ┌──────────────────────────────────────────────────┐  │  │
│  │  │ Container Registry           │  │  │
│  │  │ • Docker image storage         │  │  │
│  │  │ • Image versioning          │  │  │
│  │  └──────────────────────────────────────────────────┘  │  │
│  └──────────────────────────────────────────────────────────┘  │
│                │                │
│  ┌──────────────────────────────────────────────────────────┐  │
│  │       GitHub Actions - CI/CD Pipeline      │  │
│  │                        │  │
│  │ On Push to main:              │  │
│  │ 1. Run unit tests            │  │
│  │ 2. Run integration tests          │  │
│  │ 3. Code quality checks (pylint, black)      │  │
│  │ 4. Build Docker image           │  │
│  │ 5. Push to Container Registry        │  │
│  │ 6. Deploy to Cloud Run          │  │
│  │ 7. Run smoke tests on production       │  │
│  │ 8. Notify team (Slack)          │  │
│  └──────────────────────────────────────────────────────────┘  │
│                      │
└────────────────────────────────────────────────────────────────┘
```

5.2 Component Interaction Flow

5.2.1 Data Flow (Training Phase)

```
1. Airflow triggers daily data collection DAG
  ↓
2. Fetch data from external APIs
   ├── eBird: Bird observations
   ├── Xeno-Canto: Audio recordings
   └── Weather: Environmental data
  ↓
3. Validate data quality
   ├── Check for missing values
   ├── Verify data schemas
   └── Flag anomalies
  ↓
4. Store raw data
   ├── Audio → Cloud Storage
   ├── Metadata → PostgreSQL
   └── Version with DVC
  ↓
5. Feature extraction pipeline
   ├── Audio → MFCC, spectral features
   ├── Weather → Engineered features
   └── Combine into feature vectors
  ↓
6. Store features in Feature Store
  ↓
7. Weekly training DAG triggers
  ↓
8. Prepare training data
   ├── Temporal split (train/val/test)
   ├── Create sequences (168h windows)
   └── Normalize features
  ↓
9. Train models
   ├── LSTM stress predictor
   ├── VAE acoustic generator
   └── Risk classifier
  ↓
10. Evaluate models
    ├── Compute metrics
    ├── Generate plots
    └── Compare with current production
  ↓
11. Log to MLflow
    ├── Parameters
    ├── Metrics
    ├── Artifacts
    └── Model binary
  ↓
12. If improved:
   ├── Register new model version
   ├── Stage as "Production"
   └── Notify team
   Else:
   └── Keep current production model
```

5.2.2 Data Flow (Inference Phase)

```
1. User sends POST /predict/stress request
  ↓
2. FastAPI receives request
  ↓
3. Pydantic validates input
   ├── Check required fields
   ├── Validate data types
   └── Return 422 if invalid
  ↓
4. Extract features from input
   ├── Download audio from URL
   ├── Extract MFCC features
   ├── Format weather data
   └── Create feature vector
  ↓
5. Load production model from MLflow
  ↓
6. Run inference
   ├── Forward pass through LSTM
   ├── Get stress score + uncertainty
   └── Classify risk level
  ↓
7. Log prediction
   ├── Input features
   ├── Output prediction
   ├── Timestamp
   └── Latency
  ↓
8. Return JSON response
  {
   "stress_score": 0.81,
   "risk_level": "HIGH",
   "confidence": 0.92,
   "disaster_type_prob": {...},
   "model_version": "v3.2"
  }
  ↓
9. Background: Send metrics to Prometheus
   ├── Inference latency
   ├── Request count
   └── Prediction distribution
  ↓
10. Background: Monitor for drift (Evidently)
    ├── Compare feature distribution
   ├── Check prediction distribution
   └── Alert if drift detected
```

5.3 Database Schema Design

5.3.1 PostgreSQL Schema

Table: bird_observations

```sql
CREATE TABLE bird_observations (
    id SERIAL PRIMARY KEY,
    observation_id VARCHAR(100) UNIQUE NOT NULL,
    species_code VARCHAR(50) NOT NULL,
    species_common_name VARCHAR(200),
    location_lat DECIMAL(10, 7) NOT NULL,
    location_lng DECIMAL(10, 7) NOT NULL,
    location_name VARCHAR(200),
    observation_date TIMESTAMP NOT NULL,
    observation_count INTEGER,
    source VARCHAR(50) DEFAULT 'ebird',
    metadata JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_obs_date ON bird_observations(observation_date);
CREATE INDEX idx_obs_location ON bird_observations(location_lat, location_lng);
CREATE INDEX idx_obs_species ON bird_observations(species_code);
```

Table: audio_recordings

```sql
```

```sql
CREATE TABLE audio_recordings (
    id SERIAL PRIMARY KEY,
    recording_id VARCHAR(100) UNIQUE NOT NULL,
    species_code VARCHAR(50) NOT NULL,
    location_lat DECIMAL(10, 7),
    location_lng DECIMAL(10, 7),
    recording_date TIMESTAMP,
    duration_seconds DECIMAL(10, 2),
    file_url TEXT NOT NULL,
    local_path TEXT,
    quality_rating VARCHAR(10),
    recordist VARCHAR(200),
    source VARCHAR(50) DEFAULT 'xeno-canto',
    metadata JSONB,
    processed BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_audio_species ON audio_recordings(species_code);
CREATE INDEX idx_audio_processed ON audio_recordings(processed);
```

Table: audio_features

```
sql
```

```sql
CREATE TABLE audio_features (
    id SERIAL PRIMARY KEY,
    recording_id VARCHAR(100) REFERENCES audio_recordings(recording_id),
    -- MFCC features
    mfcc_mean FLOAT[],  -- Array of 13 values
    mfcc_std FLOAT[],
    mfcc_delta FLOAT[],
    -- Spectral features
    spectral_entropy_mean FLOAT,
    spectral_entropy_std FLOAT,
    spectral_entropy_max FLOAT,
    spectral_centroid_mean FLOAT,
    spectral_centroid_std FLOAT,
    spectral_rolloff_mean FLOAT,
    spectral_rolloff_std FLOAT,
    -- Zero crossing rate
    zcr_mean FLOAT,
    zcr_std FLOAT,
    -- Chroma features
    chroma_mean FLOAT[],  -- Array of 12 values
    -- Metadata
    extraction_version VARCHAR(20),
    extracted_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_features_recording ON audio_features(recording_id);
```

Table: weather_data

```
sql
```

```sql
CREATE TABLE weather_data (
    id SERIAL PRIMARY KEY,
    location_name VARCHAR(200),
    location_lat DECIMAL(10, 7) NOT NULL,
    location_lng DECIMAL(10, 7) NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    -- Raw weather
    pressure_hpa FLOAT,
    temperature_c FLOAT,
    humidity_percent FLOAT,
    wind_speed_kmh FLOAT,
    wind_direction_deg INTEGER,
    precipitation_mm FLOAT,
    cloud_cover_percent INTEGER,
    -- Engineered features
    pressure_drop_1h FLOAT,
    pressure_drop_6h FLOAT,
    pressure_drop_24h FLOAT,
    pressure_rolling_mean_24h FLOAT,
    pressure_rolling_std_24h FLOAT,
    temp_change_6h FLOAT,
    humidity_change_6h FLOAT,
    -- Metadata
    source VARCHAR(50),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(location_lat, location_lng, timestamp, source)
);

CREATE INDEX idx_weather_location_time ON weather_data(location_lat, location_lng, timestamp);
CREATE INDEX idx_weather_time ON weather_data(timestamp);
```

Table: disaster_events

```
sql
```

```sql
CREATE TABLE disaster_events (
    id SERIAL PRIMARY KEY,
    event_id VARCHAR(100) UNIQUE NOT NULL,
    event_type VARCHAR(50) NOT NULL,  -- earthquake, flood, storm, cyclone
    magnitude FLOAT,
    location_lat DECIMAL(10, 7) NOT NULL,
    location_lng DECIMAL(10, 7) NOT NULL,
    event_datetime TIMESTAMP NOT NULL,
    radius_affected_km FLOAT,
    severity VARCHAR(20),  -- minor, moderate, major, severe
    casualties INTEGER,
    description TEXT,
    source VARCHAR(100),
    metadata JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_disaster_time ON disaster_events(event_datetime);
CREATE INDEX idx_disaster_location ON disaster_events(location_lat, location_lng);
CREATE INDEX idx_disaster_type ON disaster_events(event_type);
```

Table: feature_vectors

```
sql
```

```sql
CREATE TABLE feature_vectors (
    id SERIAL PRIMARY KEY,
    location_lat DECIMAL(10, 7) NOT NULL,
    location_lng DECIMAL(10, 7) NOT NULL,
    timestamp TIMESTAMP NOT NULL,
    -- Feature components
    audio_features JSONB,  -- All audio features as JSON
    weather_features JSONB,  -- All weather features as JSON
    temporal_features JSONB,  -- Hour, day, cyclical encodings
    -- Combined vector
    feature_vector FLOAT[],  -- Array of all 92 features
    -- Labels (for training)
    stress_label FLOAT,  -- Ground truth stress (if available)
    disaster_occurred BOOLEAN DEFAULT FALSE,
    disaster_event_id VARCHAR(100) REFERENCES disaster_events(event_id),
    -- Metadata
    feature_version VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    UNIQUE(location_lat, location_lng, timestamp)
);

CREATE INDEX idx_features_location_time ON feature_vectors(location_lat, location_lng, timestamp);
CREATE INDEX idx_features_disaster ON feature_vectors(disaster_occurred);
```

Table: predictions

```
sql
```

```sql
CREATE TABLE predictions (
    id SERIAL PRIMARY KEY,
    prediction_id UUID DEFAULT gen_random_uuid(),
    location_lat DECIMAL(10, 7) NOT NULL,
    location_lng DECIMAL(10, 7) NOT NULL,
    prediction_timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    -- Input
    input_features JSONB,
    -- Output
    stress_score FLOAT NOT NULL,
    uncertainty FLOAT,
    risk_level VARCHAR(20),  -- LOW, MEDIUM, HIGH, CRITICAL
    disaster_type_probabilities JSONB,
    -- Model info
    model_version VARCHAR(50),
    model_id VARCHAR(100),
    -- Performance
    inference_time_ms FLOAT,
    -- Validation (if disaster actually occurred)
    actual_stress FLOAT,
    actual_disaster_occurred BOOLEAN,
    prediction_accuracy FLOAT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE INDEX idx_pred_time ON predictions(prediction_timestamp);
CREATE INDEX idx_pred_location ON predictions(location_lat, location_lng);
CREATE INDEX idx_pred_model ON predictions(model_version);
```

Table: model_registry

```
sql
```

```sql
CREATE TABLE model_registry (
    id SERIAL PRIMARY KEY,
    model_id VARCHAR(100) UNIQUE NOT NULL,
    model_name VARCHAR(200) NOT NULL,
    model_version VARCHAR(50) NOT NULL,
    model_type VARCHAR(50),  -- lstm_stress, vae_acoustic, risk_classifier
    -- Training info
    training_date TIMESTAMP,
    training_duration_minutes INTEGER,
    training_dataset_size INTEGER,
    hyperparameters JSONB,
    -- Performance
    accuracy FLOAT,
    precision FLOAT,
    recall FLOAT,
    f1_score FLOAT,
    custom_metrics JSONB,
    -- Deployment
    status VARCHAR(20),  -- staging, production, archived
    mlflow_run_id VARCHAR(100),
    artifact_path TEXT,
    -- Metadata
    created_by VARCHAR(100),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    deployed_at TIMESTAMP,
    archived_at TIMESTAMP
);

CREATE INDEX idx_model_status ON model_registry(status);
CREATE INDEX idx_model_version ON model_registry(model_version);
```

---

6. DATA REQUIREMENTS

6.1 Training Data Requirements

6.1.1 Data Volume Estimates

Minimum Viable Dataset:

- **Audio recordings:** 1,000+ samples
    - Species: Crows (primary), Sparrows, Pigeons
    - Duration: 5-30 seconds each
    - Total: ~5 hours of audio

- **Weather data:** 365 days × 24 hours = 8,760 hourly records per location
  - Locations: 5-10 monitoring sites
  - Total: 40,000-80,000 weather records
- **Disaster events:** 50-100 historical events
  - Types: Earthquakes (60%), Floods (20%), Storms (20%)
  - Geographic coverage: India (focus on seismic zones)

Optimal Dataset:

- Audio recordings: 5,000+ samples
- Weather data: 2-3 years historical (17,520-26,280 records/location)
- Disaster events: 200+ events

6.1.2 Data Quality Requirements

Audio Quality:

- Sample rate: ≥22,050 Hz
- Bit depth: 16-bit minimum
- File format: WAV or high-quality MP3 (≥192 kbps)
- Background noise: Minimal (prefer quality rating "A" from Xeno-Canto)
- Duration: 5-30 seconds (optimal)

Weather Data Quality:

- Missing values: <5% per feature
- Temporal resolution: Hourly
- Sensor accuracy: Standard meteorological grade
- Outlier detection: Automated flagging of anomalous values

Disaster Event Data Quality:

- Precise timestamp (hour-level accuracy minimum)
- Accurate geolocation (GPS coordinates)
- Verified magnitude/severity
- Source credibility (USGS, official agencies)

6.1.3 Data Labeling Strategy

Stress Labels: Since direct stress measurement is impossible, labels are derived:

1. **Disaster-based labeling:**
    - 72-48h before disaster: Stress = 0.3-0.5 (moderate)
    - 48-24h before disaster: Stress = 0.5-0.7 (high)
    - 24-0h before disaster: Stress = 0.7-1.0 (critical)
    - Normal times (>72h from disaster): Stress = 0.0-0.3 (low)
2. **Weather-based augmentation:**
    - Rapid pressure drops (>5 hPa in 6h): +0.1 stress
    - Extreme weather conditions: +0.1 stress
    - Normal conditions: No adjustment
3. **Expert annotation (optional):**
    - Manual review of acoustic patterns
    - Validation of auto-generated labels

Quality Assurance:

- Cross-validation with multiple disaster databases
- Temporal alignment verification (disaster time ±1 hour tolerance)
- Geographic matching (disaster location within 100km of observation)

6.2 Data Collection Plan

6.2.1 Initial Data Collection (Week 1, Days 1-2)

Day 1: Historical Disaster Data

```
Step 1: USGS Earthquake Database
- Download earthquakes: magnitude ≥4.0, India, 2020-2024
- Format: CSV with timestamp, lat/lng, magnitude
- Expected: 200-300 events

Step 2: India Disaster Database
- Scrape floods and storms from IMD archives
- Manual compilation from news sources (if needed)
- Format: Same as earthquake data

Step 3: Data cleaning
- Remove duplicates
- Standardize formats
- Verify coordinates
```

Day 2: Historical Weather Data

```
Step 1: Define monitoring locations
- Identify 5-10 cities in seismic zones
- Cities: Delhi, Ahmedabad, Mumbai, Guwahati, Srinagar

Step 2: Download OpenWeather historical data
- Use Historical Weather API
- Date range: 2 years (2022-2024)
- Parameters: All weather features listed
- Format: JSON → Convert to CSV

Step 3: Data validation
- Check for gaps
- Interpolate missing values (max 2-hour gaps)
- Flag quality issues
```

6.2.2 Audio Data Collection (Week 1, Days 3-5)

```
Day 3-4: Xeno-Canto Download
- Query: Corvus splendens (House Crow), India
- Filter: Quality A, recorded after 2020
- Download: Top 1000 recordings
- Store: Cloud Storage with metadata

Day 5: Audio preprocessing
- Convert all to WAV, 22050 Hz
- Trim silence
- Normalize amplitude
- Visual inspection (sample 50 recordings)
```

6.2.3 Ongoing Data Collection (Post-deployment)

Daily Airflow DAG:

- 2 AM: Fetch previous day's eBird observations
- 2:30 AM: Download new Xeno-Canto recordings (if any)
- 3 AM: Pull last 24h weather data
- 3:30 AM: Check for disaster events (USGS real-time feeds)
- 4 AM: Data validation and storage

6.3 Data Storage & Versioning

6.3.1 Data Organization Structure

```
/data
├── raw/
│   ├── audio/
│   │   ├── 2024/
│   │   │   ├── 01/
│   │   │   │   ├── XC123456.wav
│   │   │   │   └── ...
│   │   │   └── ...
│   │   └── ...
│   ├── observations/
│   │   └── ebird_observations.csv
│   ├── weather/
│   │   └── weather_2020_2024.csv
│   └── disasters/
│       └── disaster_events.csv
│
├── processed/
│   ├── audio_features/
│   │   └── audio_features.parquet
│   ├── weather_features/
│   │   └── weather_engineered.parquet
│   └── feature_vectors/
│       ├── train.parquet
│       ├── val.parquet
│       └── test.parquet
│
├── models/
│   ├── lstm_stress_v1/
│   │   ├── model.pth
│   │   ├── config.yaml
│   │   └── metrics.json
│   └── ...
│
└── reports/
    ├── eda/
    ├── model_evaluation/
    └── monitoring/
```

6.3.2 DVC Configuration

Initialize DVC:

```bash
```

```
cd project_root
dvc init
dvc remote add -d gcs gs://bird-stress-twin-data
```

Track large files:

```bash
dvc add data/raw/audio
dvc add data/raw/weather
dvc add data/processed
dvc push
```

Version control:

```bash
git add data/raw/audio.dvc data/raw/weather.dvc
git commit -m "Add v1 training data"
git tag -a "data-v1.0" -m "Initial training dataset"
```

6.4 Data Privacy & Ethics

Privacy Considerations:

- No personally identifiable information (PII) in datasets
- Audio recordings are from public sources (Xeno-Canto)
- Bird observations are publicly available (eBird)
- Weather data is non-sensitive public information

Ethical Considerations:

- Wildlife observations used responsibly (no harassment)
- Research purposes only (no commercial exploitation)
- Open data policy (share findings with scientific community)
- Acknowledge data sources in publications

Compliance:

- GDPR: Not applicable (no EU personal data)
- Data licensing: Respect Creative Commons licenses on Xeno-Canto
- Attribution: Cite all data sources properly

7. FEATURE ENGINEERING SPECIFICATIONS

7.1 Audio Feature Extraction

7.1.1 MFCC (Mel Frequency Cepstral Coefficients)

Mathematical Background: MFCCs represent the short-term power spectrum of a sound, modeling the human auditory system's response.

Extraction Process:

python

```python
import librosa
import numpy as np

def extract_mfcc_features(audio_path, sr=22050, n_mfcc=13):
    """
    Extract MFCC features from audio file.

    Parameters:
    -----------
    audio_path : str
        Path to audio file
    sr : int
        Target sampling rate (22050 Hz optimal for bird calls)
    n_mfcc : int
        Number of MFCC coefficients (13 is standard)

    Returns:
    --------
    dict : MFCC features (mean, std, delta)
    """
    # Load audio
    y, sr = librosa.load(audio_path, sr=sr)

    # Extract MFCCs
    mfccs = librosa.feature.mfcc(
        y=y,
        sr=sr,
        n_mfcc=n_mfcc,
        n_fft=2048,       # Frame size: ~93ms at 22050 Hz
        hop_length=512,  # Hop size: ~23ms (75% overlap)
        n_mels=128       # Number of Mel bands
    )

    # Compute statistics across time
    mfcc_mean = np.mean(mfccs, axis=1)  # Shape: (13,)
    mfcc_std = np.std(mfccs, axis=1)    # Shape: (13,)

    # First-order derivatives (velocity)
    mfcc_delta = librosa.feature.delta(mfccs)
    mfcc_delta_mean = np.mean(mfcc_delta, axis=1)  # Shape: (13,)

    return {
        'mfcc_mean': mfcc_mean.tolist(),         # 13 features
        'mfcc_std': mfcc_std.tolist(),           # 13 features
        'mfcc_delta_mean': mfcc_delta_mean.tolist()  # 13 features
    }  # Total: 39 MFCC features
```

Interpretation:

- **MFCC 1-13 (mean):** Represent spectral envelope (vocal tract shape)
- **Lower coefficients (1-5):** Capture broad spectral shape
- **Higher coefficients (6-13):** Capture finer spectral details
- **Std deviation:** Indicates variability (stressed birds may have more erratic calls)
- **Delta coefficients:** Capture temporal dynamics (rate of vocal change)

7.1.2 Spectral Entropy

Mathematical Formulation:$H = -\sum_{i=1}^{N} p_i \log_2(p_i)$

Where p_i is the normalized power in frequency bin i.

Implementation:

```
python
```

```
def extract_spectral_entropy(audio_path, sr=22050):
    """
    Calculate spectral entropy - measure of spectral complexity.
    Low entropy = tonal (pure frequencies)
    High entropy = noisy (spread across frequencies)

    Stressed birds tend to produce more chaotic (higher entropy) calls.
    """
    y, sr = librosa.load(audio_path, sr=sr)

    # Compute magnitude spectrogram
    spec = np.abs(librosa.stft──────────────────────────┬──────────────────
                         ↓
┌─────────────────────────────────────────────────────────────────────┐
│          FEATURE ENGINEERING LAYER                    │
├─────────────────────────────────────────────────────────────────────┤
│                                       │
│  ┌──────────────────────┐  ┌────────────────────────┐        │
│  │ Audio Processing    │  │ Weather Processing  │        │
│  │ ─────────────────── │  │ ──────────────────  │        │
│  │ • Load audio       │  │ • Load time series │        │
│  │ • Denoise          │  │ • Compute lags     │        │
│  │ • Extract MFCCs    │  │ • Rolling stats    │        │
│  │ • Spectral features │  │ • Gradients        │        │
│  │ • Statistical agg.  │  │ • Cyclical encoding │        │
│  └──────────┬───────────┘  └───────────┬────────────┘        │
│       │           │           │
│       └───────────┬───────────────┘           │
│           ↓                     │
│  ┌──────────────────────────────────────────────────────────┐ │
│  │       Feature Store (PostgreSQL)           │ │
│  │ • Engineered features table               │ │
│  │ • Feature metadata & statistics            │ │
│  │ • Temporal indexing for time-series         │ │
│  └──────────────────────────────────────────────────────────┘ │
│                                       │
└──────────────────────────────────┬──────────────────────────────────┘
                         ↓
┌─────────────────────────────────────────────────────────────────────┐
│            ML TRAINING LAYER                  │
├─────────────────────────────────────────────────────────────────────┤
│                                       │
│  ┌──────────────────────────────────────────────────────────┐ │
│  │          Model Training Pipeline            │ │
│  │                               │ │
│  │ 1. Data Preparation                    │ │
│  │     ├── Temporal train/val/test split              │  │
│  │     ├── Feature normalization                      │  │
│  │     └── Sequence creation (168h windows)           │  │
│  │                                                    │  │
│  │  2. Model Training                                 │  │
│  │     ├── LSTM Stress Predictor                      │  │
│  │     ├── VAE Acoustic Generator                     │  │
│  │     └── Risk Classifier                            │  │
│  │                                                    │  │
│  │  3. Model Evaluation                               │  │
│  │     ├── Accuracy, Precision, Recall                │  │
│  │     ├── Confusion matrix analysis                  │  │
│  │     └── Lead time validation                       │  │
│  │                                                    │  │
│  │  4. Experiment Tracking (MLflow)                   │  │
│  │     ├── Log parameters                             │  │
│  │     ├── Log metrics                                │  │
│  │     ├── Save artifacts (model, plots)              │  │
│  │     └── Tag experiments                            │  │
│  └────────────────────────────────────────────────────┘  │
│                  │                       │                │
│                  ↓                       │                │
│  ┌────────────────────────────────────────────────────┐  │
│  │            MLflow Model Registry                   │  │
│  │  • Model versioning (v1, v2, v3...)                │  │
│  │  • Model staging (Staging → Production)            │  │
│  │  • Model lineage tracking                          │  │
│  │  • Model comparison dashboard                      │  │
│  └────────────────────────────────────────────────────┘  │
│                                          │                │
└
```