# **ArduWeatherX: Real-Time IoT Weather Monitoring System**

## **Project Overview**

ArduWeatherX is a comprehensive, end-to-end IoT solution designed to monitor environmental data in real-time. The system addresses the challenge of how to easily collect data from remote sensors and visualize it on an accessible, modern web platform.1

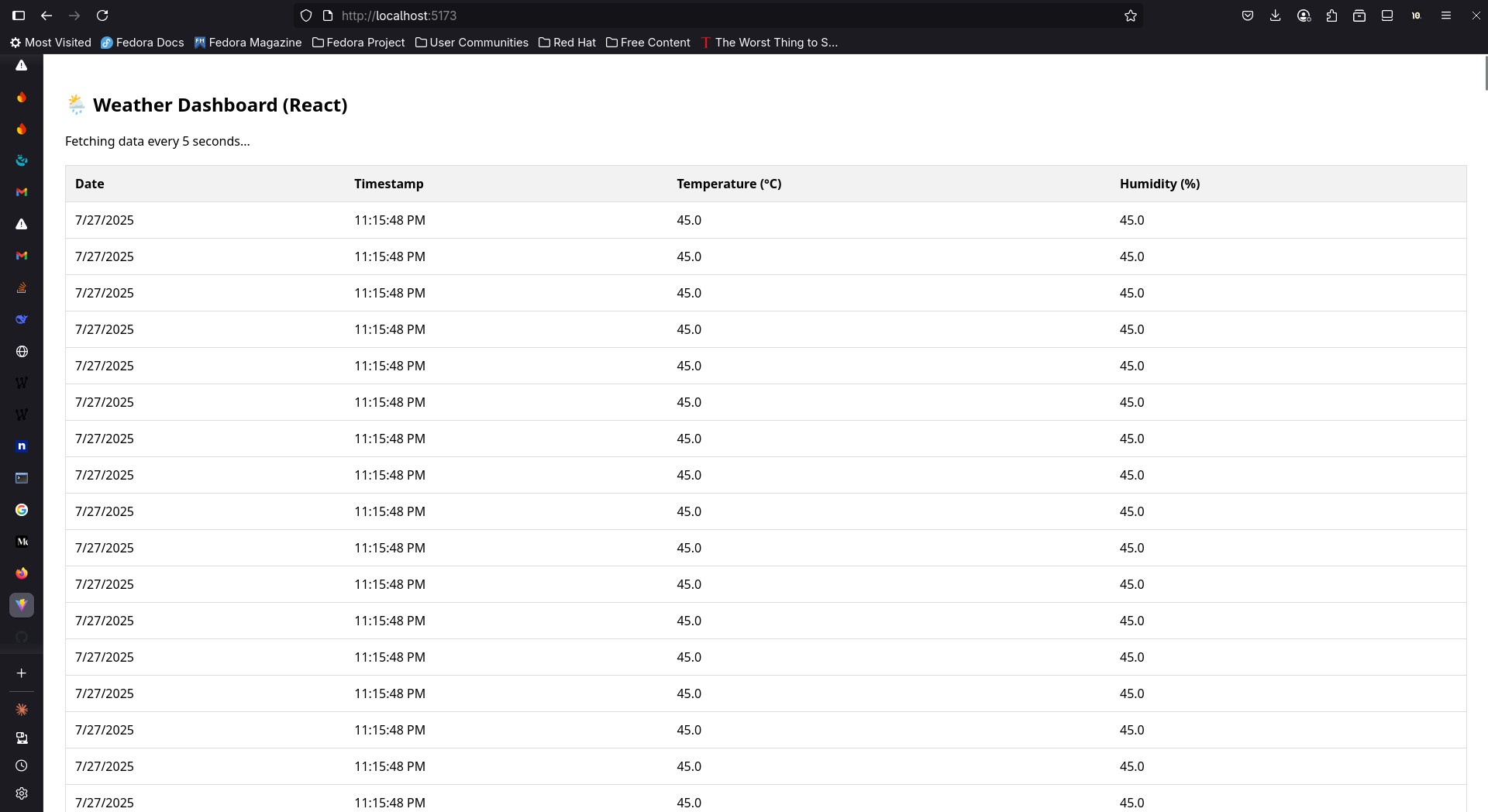
It provides a complete, scalable, and cost-effective system by leveraging a simulated IoT sensor, a cloud-native backend, a real-time database, and an interactive web dashboard to deliver up-to-the-second weather data.1 This project serves as a robust template for various real-world applications, including smart agriculture, home automation, and environmental studies.

## **🚀 Live Demo**

A live version of the deployed application can be accessed at the following link.

[**https://arduweatherx.yourdomain.com**](https://arduweatherx.yourdomain.com/) *(Note: Replace with your actual deployment URL)*

## **📸 Dashboard Preview**

****

## **Core Features**

* **Real-time Sensor Simulation**: Utilizes the Wokwi platform to simulate a DHT11 temperature and humidity sensor, providing a continuous, realistic stream of data for development and testing without requiring physical hardware. The simulation includes a 5% chance of generating invalid readings to allow for robust error-handling development.1
* **Scalable Serverless Backend**: Transitions seamlessly from a local Flask server for rapid development to a highly scalable, event-driven AWS Lambda backend for production. This serverless architecture ensures high availability and cost efficiency through a pay-per-use model.1
* **Persistent Cloud Storage**: Leverages Firebase Realtime Database for low-latency, persistent storage of all incoming sensor data. The data is structured and indexed by timestamp for efficient querying and real-time synchronization with the frontend.1
* **Interactive Web Dashboard**: A modern, responsive frontend built with React and Vite, providing a real-time visualization of temperature and humidity data. The dashboard updates automatically as new data arrives in Firebase, without requiring page reloads.1
* **Secure & Efficient Delivery**: In the production environment, the frontend is deployed to Amazon S3 and served globally via the AWS CloudFront Content Delivery Network (CDN). This ensures low latency for users worldwide and a secure connection via HTTPS.1
* **Automated CI/CD Pipeline**: Features a complete Continuous Integration and Continuous Deployment (CI/CD) pipeline using AWS CodePipeline. This pipeline automates the build, test, and deployment process for both the backend Lambda function and the frontend React application, ensuring rapid and reliable updates.1

## **System Architecture and Data Flow**

The project is designed with two distinct architectural models: one optimized for local development and another for scalable production deployment on AWS. This separation is a hallmark of professional software engineering, allowing for rapid iteration locally while ensuring robustness in production. The local setup prioritizes developer velocity with tools like hot-reloading, whereas the cloud architecture prioritizes scalability, reliability, and security.1

### **Local Development Architecture**

This setup is optimized for rapid development, hot-reloading, and easy debugging of individual components.

*Data Flow:* Wokwi Simulator → Local Flask API (localhost:5000) → Firebase Realtime DB → React Dev Server (localhost:5173) → Browser

### **Production AWS Architecture**

This serverless architecture is designed for scalability, reliability, and operational excellence, decoupling components and minimizing infrastructure management.

Backend Data Flow: Wokwi Simulator → AWS API Gateway Endpoint → AWS Lambda Function → Firebase Realtime DB

Frontend Data Flow: User's Browser → AWS CloudFront CDN → Amazon S3 Bucket → User's Browser

The following diagram illustrates the production architecture:

Code snippet

graph TD  
 subgraph "Sensor Simulation"  
 A  
 end  
 subgraph "AWS Backend"  
 B(AWS API Gateway)  
 C{AWS Lambda Function}  
 end  
 subgraph "Database"  
 D  
 end  
 subgraph "AWS Frontend Hosting"  
 F(AWS CloudFront CDN)  
 G  
 end  
 subgraph "User"  
 E  
 H{React Frontend UI}  
 end  
  
 A -->|1. HTTPS POST Request| B;  
 B -->|2. Triggers| C;  
 C -->|3. Writes Data| D;  
 E -->|4. Requests Website| F;  
 F -->|5. Serves Static Files| G;  
 G -->|6. Loads App| H;  
 D -.->|7. Real-time Sync| H;  
 H -->|Displays Data| E;

## **Technology Stack**

The project utilizes a modern, robust technology stack chosen to support real-time data processing, scalability, and a rich user experience.1

|  |  |  |
| --- | --- | --- |
| Category | Technology | Purpose |
| **Frontend** | React, Vite, TailwindCSS | Building a dynamic, responsive, and real-time user interface. |
| **Backend** | Python, Flask, AWS Lambda | Data ingestion, validation, and processing. Flask for local dev, Lambda for prod. |
| **Database** | Firebase Realtime DB | NoSQL, real-time, persistent cloud storage for sensor data. |
| **IoT Simulation** | Wokwi (Arduino C++) | Simulating a DHT11 sensor to generate temperature/humidity data. |
| **Cloud & DevOps** | AWS (S3, CloudFront, API Gateway, Lambda, CodePipeline) | Hosting, deployment, CI/CD, and serverless infrastructure. |
| **Tooling** | Node.js, ESLint, Git, Terraform | Frontend runtime, code linting, version control, Infrastructure as Code. |

## **Getting Started: Local Development Environment**

Follow these steps to set up and run the project on your local machine for development and testing.1

### **Prerequisites**

Ensure you have the following software installed on your system:

* Node.js (≥16) 1
* Python (≥3.9) 1
* Git version control

### **1. Clone the Repository**

Bash

git clone https://github.com/yourusername/ArduWeatherX.git  
cd ArduWeatherX

### **2. Configure Environment**

Before running the application, you must configure the backend with your Firebase credentials.

* Follow the instructions in the **(#configuration-and-security)** section below to set up your backend/.env and backend/serviceAccountKey.json files. This step is mandatory for the application to connect to Firebase.

### **3. Backend Setup (Flask)**

In a terminal, navigate to the backend directory to set up the Python environment.

Bash

cd backend  
  
# Create a virtual environment  
python -m venv venv  
  
# Activate the virtual environment  
# On Windows:  
venv\Scripts\activate  
# On macOS/Linux:  
source venv/bin/activate  
  
# Install dependencies  
pip install -r requirements.txt  
  
# Run the Flask server  
# The server will start on http://localhost:5000  
flask run

### **4. Frontend Setup (React)**

Open a **new terminal** and navigate to the frontend directory.

Bash

cd frontend  
  
# Install dependencies  
npm install  
  
# Start the Vite development server  
# The application will be available at http://localhost:5173  
npm run dev

### **5. Wokwi Simulation Setup**

1. Open the(<https://wokwi.com/projects/new/arduino-uno>).
2. Copy the code from the /wokwi/arduino\_dht11.ino file in this repository and paste it into the Wokwi editor.
3. You will need to configure the simulation to send data to your local backend. This typically involves modifying the Arduino code to make an HTTP POST request to http://localhost:5000/data. Since Wokwi runs in the browser, you may need a tool like ngrok to expose your local server to the internet so Wokwi can reach it.

### **6. Running the Full Stack with VS Code**

This repository is configured for a streamlined experience in Visual Studio Code.

1. Install the recommended extensions when prompted by VS Code.
2. Open the Command Palette (Ctrl+Shift+P), select **Debug: Select and Start Debugging**.
3. Choose the **Full Stack** compound configuration from the dropdown menu. This will launch both the Python Flask backend and the React frontend development server simultaneously.1

## **Configuration and Security**

Proper configuration is crucial for the application's functionality and security. This section details the setup of environment variables and Firebase integration.1

### **Backend Environment Variables (.env)**

The backend uses a .env file to manage configuration for the Flask application.

1. In the backend directory, create a file named .env.
2. Add the following content:  
   FLASK\_APP=backend.py  
   FLASK\_ENV=development

**Important**: The .env file should be listed in your .gitignore file and should never be committed to version control.

### **Firebase Integration**

#### **Service Account Key (serviceAccountKey.json)**

The backend authenticates to Firebase using a service account key, which grants privileged access to your Firebase project's resources.

1. Go to your Firebase project console.
2. Navigate to **Project settings** (gear icon) > **Service accounts**.
3. Click **Generate new private key** to download your serviceAccountKey.json file.1
4. Place this file inside the backend directory.

**Warning**: Your serviceAccountKey.json file contains sensitive credentials. **DO NOT commit this file to Git.** Ensure it is included in your .gitignore file.

#### **Database Security Rules**

To control data access, you must configure security rules in your Firebase Realtime Database. The following rules allow public read access (for the frontend) but restrict write access to authenticated sources (your backend).1

1. In the Firebase console, go to **Realtime Database** > **Rules**.
2. Replace the default rules with the following:  
   JSON  
   {  
    "rules": {  
    "weather": {  
    ".read": true,  
    ".write": "auth!= null",  
    ".indexOn": ["timestamp"]  
    }  
    }  
   }  
     
   The .indexOn rule is critical for optimizing queries that sort or filter data by the timestamp field.1

#### **Security Note on the Public API Endpoint**

The Firebase rule ".write": "auth!= null" is enforced because the firebase-admin SDK used in the backend authenticates using the service account key. However, the API endpoint itself (/data) is publicly exposed. In a production scenario, it is highly recommended to secure this endpoint to prevent unauthorized data submission. On AWS, this can be achieved by configuring the API Gateway to require an API Key and setting up a Usage Plan. The client (Wokwi simulator or a real IoT device) would then need to include this API key in its request headers.

## **API Reference**

The backend provides the following API endpoints for interacting with the weather data.1

|  |  |  |  |
| --- | --- | --- | --- |
| Method | Route | Description | Request Body (JSON) |
| POST | /data | Submits new sensor data from the IoT device/simulator. The backend adds a server-side timestamp upon receipt. | { "temperature": float, "humidity": float } |
| GET | /firebase | Retrieves all stored weather data from the Firebase Realtime Database. | (None) |

## **Deployment to AWS**

The project is designed for a robust, serverless deployment on AWS. This approach ensures scalability, high availability, and cost-effectiveness. The detailed deployment plan involves several phases and leverages a suite of AWS services.1

### **Core AWS Services Used**

* **AWS Lambda**: Runs the Python backend code without provisioning or managing servers.
* **AWS API Gateway**: Creates and manages the REST API endpoint that triggers the Lambda function.
* **Amazon S3**: Stores the static files (HTML, CSS, JS) for the React frontend.
* **AWS CloudFront**: A global CDN that securely delivers the frontend content with low latency.
* **AWS CodePipeline**: Automates the CI/CD workflow, from code commit to deployment.

### **Deployment Phases Summary**

1. **Infrastructure Setup**: Provision all necessary AWS resources (Lambda, API Gateway, S3, etc.) using an Infrastructure as Code tool like Terraform or AWS CloudFormation.
2. **Backend Deployment**: Package the Python application and its dependencies into a deployment zip file and upload it to AWS Lambda.
3. **Frontend Deployment**: Build the production version of the React application (npm run build) and sync the resulting static files to the designated S3 bucket.
4. **CI/CD Configuration**: Set up AWS CodePipeline to monitor the GitHub repository. On every push to the main branch, the pipeline automatically builds the artifacts and deploys the new versions of the backend and frontend.

For a complete step-by-step guide, refer to the detailed roadmap in the project documentation.1

## **Project Structure**

The repository is organized into distinct directories for the frontend, backend, and configuration files, promoting a clean separation of concerns.1

arduweatherx/  
├── backend/  
│ ├──.env # Environment variables (local only, not in Git)  
│ ├── backend.py # Flask/Lambda application logic  
│ ├── requirements.txt # Python dependencies  
│ └── serviceAccountKey.json # Firebase credentials (local only, not in Git)  
├── frontend/  
│ ├── public/ # Static assets for Vite  
│ ├── src/ # React source code  
│ │ ├── App.jsx  
│ │ └── DataTable.jsx  
│ ├── package.json # Frontend dependencies and scripts  
│ └── vite.config.js # Vite configuration  
├──.vscode/  
│ ├── launch.json # VS Code debugger configurations  
│ └── settings.json # Workspace settings  
├──.gitignore # Specifies files and directories to be ignored by Git  
└── README.md # This file

## **🤝 Contributing**

Contributions are welcome! If you have suggestions for improvements or want to report a bug, please feel free to open an issue or submit a pull request.

1. Fork the Project
2. Create your Feature Branch (git checkout -b feature/AmazingFeature)
3. Commit your Changes (git commit -m 'Add some AmazingFeature')
4. Push to the Branch (git push origin feature/AmazingFeature)
5. Open a Pull Request

## **📜 License**

This project is licensed under the MIT License. See the LICENSE file for more details.

#### Works cited

1. Data Scientist roadmap (1).pdf