FH University of Applied Sciences TECHNIKUM WIEN

The Weather Archive - Arc42 Documentation

David Veigel

, January 16, 2026

Contents

1 Introduction and Goals

The “The Weather Archive” project is a semester project for the Serverless Computing course at FH Technikum Wien. The objective is to design and implement a serverless web application that archives and displays historical weather data captured from webcams located in major European cities.

The system is designed to handle automated data ingestion, image processing, and video generation in a scalable and cost-effective manner using cloud-native technologies.

1.1 Requirements Overview

- **Automated Image Archiving**: Securely receive and store images from distributed webcams.
- **Image Processing**: Automatically compress and optimize images upon upload (PictureService).
- **Video Generation**: Periodically compile archived images into time-lapse videos (VideoService).
- **Metadata Visualization**: Allow users to filter and view weather data by city and date.
- **Performance**: Utilize caching strategies (Redis) to minimize database load and ensure low latency.

1.2 Quality Goals

1. **Scalability**: The system must handle variable loads, from sporadic uploads to high read traffic.
2. **Cost Efficiency**: Leverage a serverless "pay-as-you-go" model to minimize idle resource costs.
3. **Availability**: Ensure the application is accessible and resilient to failures.

2 Context and Scope

This section describes the boundaries of the Weather Archive system and its interactions with external users and systems.

2.1 Business Context

The system interacts with two primary actors:

- **Webcams (Technical User)**: Distributed IoT devices or scripts that capture weather images. They interact with the system via a secure API using an API Key. They are responsible for uploading raw image data.
- **Web Users (Human)**: End-users who access the web interface to view archived images, generated videos, and meteorological metadata (cities, dates).

Basic Interaction Flow:

- **Webcams** $\rightarrow$ Authenticate (API Key) $\rightarrow$ Upload Image.
- **Users** $\rightarrow$ Request Data $\rightarrow$ View Images/Videos.

3 Solution Strategy

The solution aims to build a fully serverless architecture on AWS (Amazon Web Services), minimizing operational overhead while maximizing scalability. The strategy relies on event-driven computing and managed services.

3.1 Key Technology Decisions

- **Compute**: AWS Lambda is used for all backend logic (API handlers, background processing). This eliminates the need for managing virtual machines (EC2).
- **Storage**:
  - **Amazon S3**: Used for storing unstructured blob data (Raw Images, Processed Images, Videos). It provides high durability and triggers events on file upload.

  - **Amazon RDS (PostgreSQL)**: Used for structured relational data (metadata, file paths, timestamps).
  - **Redis (Cloud)**: Implements the Cache-Aside pattern to reduce read latency and database connections.
- **API Management**: AWS API Gateway (HTTP API) serves as the entry point, routing requests to appropriate Lambda functions.
- **Infrastructure as Code**: Terraform is used to provision and manage all resources, ensuring reproducibility.

4 Building Block View

4.1 Level 1: Whitebox Overall System

The system acts as a central archive receiving data from webcams and serving content to a frontend application.

Webcam Pushes images to the Weather Archive via API.

Weather Archive System The core boundary containing all application logic and state.

User Consumes content via a web browser.

4.2 Level 2: Container View / AWS Resources

Based on the Terraform analysis, the specific AWS components are decomposed as follows:

4.2.1 API Layer (AWS API Gateway)

Resource: `aws_apigatewayv2_api.weather_api`

- Routes HTTP requests to backend Lambda functions.
- Endpoints:
  - `GET /upload-url` → `upload_service`
  - `GET /cities` → `city_service`
  - `GET /dates` → `date_service`

  - `GET /data` → `read_service`
  - `POST /video/trigger` → `test_trigger_service`

4.2.2 Compute Layer (AWS Lambda)

- **UploadService** (`weather-archive-upload`): Generates presigned S3 URLs for secure image uploads. Validates API Key.
- **PictureService** (`weather-archive-picture`): Triggered asynchronously by S3 `ObjectCreated` events. Resizes images and updates the database.
- **VideoService** (`weather-archive-video`): Scheduled by Amazon CloudWatch (`rate(24 hours)`) to compile daily time-lapse videos using an FFMPEG layer.
- **ReadService/CityService/DateService**: serve metadata to the frontend, implementing the Redis cache-aside pattern.

4.2.3 Data Layer

- **Amazon S3 Buckets**:
  - `weather-archive-raw-*` (Ingest)
  - `weather-archive-processed-*` (Optimized images)
  - `weather-archive-videos-*` (Generated time-lapses)
- **Amazon RDS**: PostgreSQL instance (`db.t3.micro`) storing `weather_archive` metadata. Securely accessed via VPC Security Groups.
- **Redis**: External Redis Cloud instance used to cache frequent query results (e.g., list of cities or available dates), bypassing the RDS instance when data is available.

5 Architecture Decisions

This section documents the rationale behind key architectural choices.

5.1 ADR-1: Serverless Compute (AWS Lambda) vs. EC2

Context: The application handles sporadic uploads from webcams and periodic video generation. Use traffic may be bursty.

Decision: Use AWS Lambda.
Justification:

- **Cost**: Lambda limits costs to execution time (milliseconds). EC2 instances would incur costs 24/7 even when idle (which is most of the time for this project).
- **Scaling**: Lambda scales automatically with the number of incoming requests (up to concurrency limits), whereas EC2 requires complex Auto Scaling Group configurations.
- **Maintenance**: Removes the need to patch OS or manage servers.

5.2 ADR-2: Relational Database (RDS) for Metadata

Context: The system needs to allow filtering by city, date, and time. Data is structured.
Decision: Amazon RDS (PostgreSQL).
Justification: Structured query capabilities (SQL) are ideal for filtering and aggregation (e.g., “Select all images for Vienna on 2024-01-01”).

5.3 ADR-3: Caching Strategy (Redis Cache-Aside)

Context: The metadata (list of cities, historical dates) changes infrequently but is read often by every user visiting the site.
Decision: Implement Redis with Cache-Aside pattern.
Justification:

- **Performance**: In-memory retrieval from Redis is significantly faster than disk-based SQL queries.
- **Database Load**: Reduces the IOPS and connection strain on the `db.t3.micro` instance.
- **Logic**: The application first checks Redis. If content is missing (cache miss), it queries RDS and populates Redis for subsequent requests.

5.4 ADR-4: Asynchronous Image Processing (S3 Triggers)

Context: High-resolution images need compression. Doing this synchronously during upload would cause timeout risks and bad user experience for the webcam API.
Decision: Use S3 Event Notifications to trigger `PictureService`.
Justification: Decouples upload from processing. The webcam gets a fast "Success" response after upload, while the heavy lifting happens in the background.