The Weather Archive - Arc42 Documentation

David Veigel

Vienna, January 19, 2026

Contents

1 Introduction and Goals

The “The Weather Archive” project is a comprehensive serverless web application developed as part of the Serverless Computing course at FH Technikum Wien. As a semester project, it aims to demonstrate the practical application of cloud-native technologies to solve a real-world data archival problem.

The primary objective is to design and implement a scalable system that ingests, archives, and displays historical weather data captured from webcams located in major European cities. By leveraging a fully serverless architecture on AWS, the project highlights the benefits of event-driven design, including automated data ingestion, on-demand image processing, and cost-effective scaling.

1.1 Requirements Overview

- **Automated Image Archiving**: The system must securely receive and store images from distributed webcams via a RESTful API.
- **Event-Driven Image Processing**: Uploaded images must trigger an immediate optimization process (compression and metadata extraction) without manual intervention (PictureService).
- **Time-Lapse Video Generation**: A scheduled process must periodically compile archived daily imagery into time-lapse videos (VideoService).
- **Metadata Visualization & Search**: Users must be able to search webcams by city (with autocomplete) and filter content by date.
- **High Performance**: The system should utilize caching strategies (Redis) to ensure low-latency access to frequently requested metadata (e.g., city lists).

1.2 Quality Goals

1. **Scalability**: The architecture must automatically adjust to variable workloads, efficiently handling bursts of webcam uploads and concurrent user access.
2. **Cost Efficiency**: The solution utilizes a "pay-as-you-go" serverless model (AWS Lambda, S3), ensuring that costs are incurred only during execution time, minimizing idle resource expenses.

3. **Availability**: The system leverages managed services to ensure high availability and fault tolerance across its components.

2 Context and Scope

This section defines the boundaries of the Weather Archive system, specifying its relationships with external actors and technical interfaces.

2.1 Business Context

The system serves two distinct user groups with specific interaction patterns:

- **Webcams (Technical User)**: These are autonomous IoT devices or scripts responsible for capturing and uploading weather imagery.
  - *Interface*: RESTful HTTP API.
  - *Authentication*: Secure API Key passed via the `x-api-key` header.
  - *Constraint*: These clients are simple and stateless; they require a straightforward upload URL mechanism.
- **Web Users (Human)**: End-users accessing the application via a modern web browser.
  - *Goal*: To explore historical weather data, visualize trends via interactive UI, and view generated time-lapse videos.
  - *Expectation*: Fast load times and responsive search mechanisms.

Basic Interaction Flow:

1. **Ingestion**: Webcams $\xrightarrow{\text{Auth}}$ API Gateway $\xrightarrow{\text{Upload}}$ S3 Bucket.
2. **Consumption**: Users $\xrightarrow{\text{HTTPS}}$ CloudFront/S3 (Frontend) $\xrightarrow{\text{API}}$ Lambda/Redis (Backend).

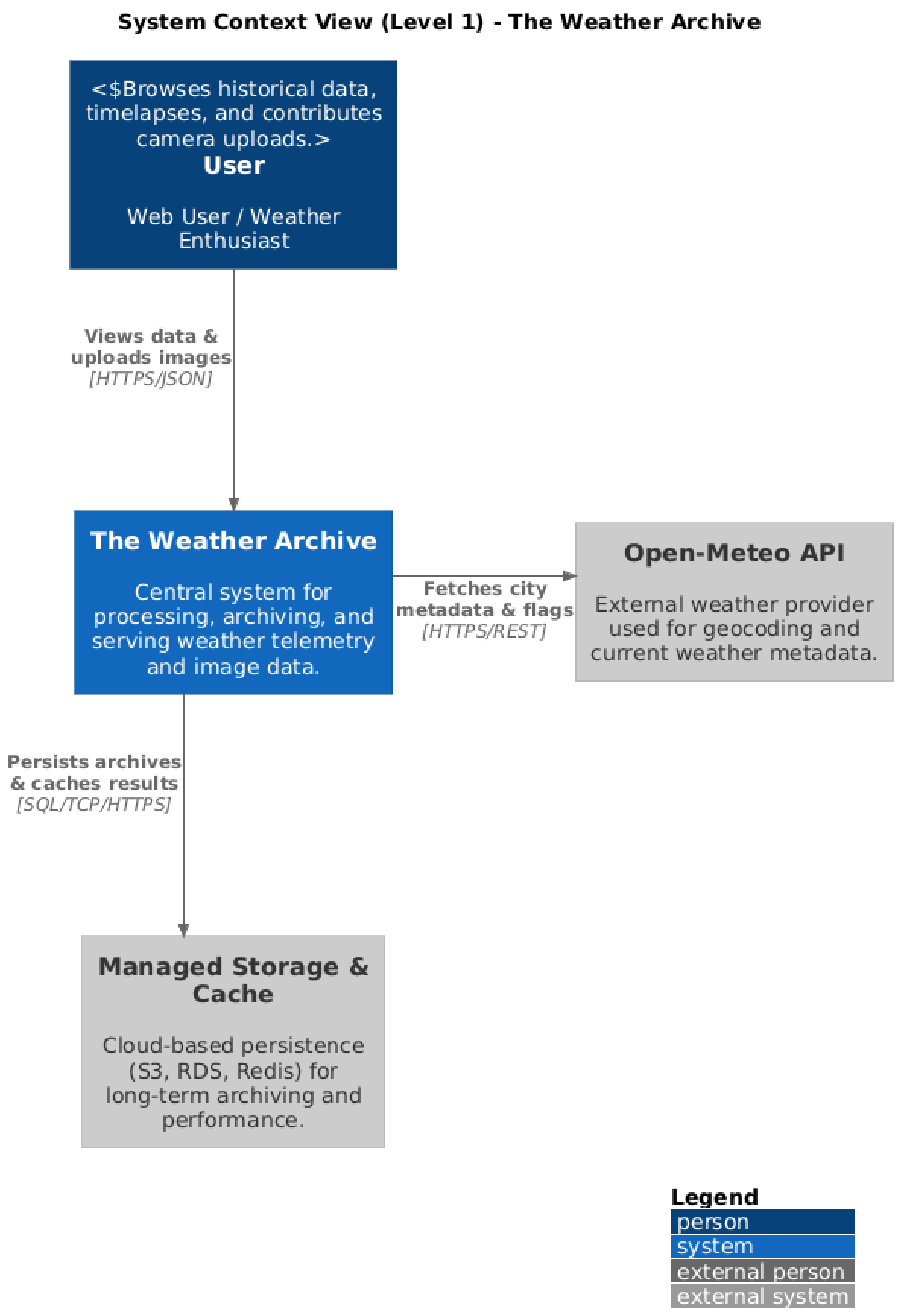
System Context View (Level 1) - The Weather Archive
<$Browses historical data, timelapses, and contributes camera uploads.>
User
Web User / Weather Enthusiast
Views data & uploads images
[HTTPS/JSON]
The Weather Archive
Central system for processing, archiving, and serving weather telemetry and image data.
Fetches city metadata & flags
[HTTPS/REST]
Open-Meteo API
External weather provider used for geocoding and current weather metadata.
Persists archives & caches results
[SQL/TCP/HTTPS]
Managed Storage & Cache
Cloud-based persistence (S3, RDS, Redis) for long-term archiving and performance.
Legend
person
system
external person
external system

Figure 1: System Context View (Level 1)

3 Solution Strategy

This section outlines the fundamental architectural and technical decisions that shape the solution. It details the reasoning behind choosing a serverless approach and provides a roadmap of the key technologies employed.

The architectural strategy focuses on a **Cloud-Native Serverless** approach on Amazon Web Services (AWS). This decision was driven by the need to minimize operational overhead (NoOps) while ensuring that the system can scale to meet the demands of sporadic image uploads and variable user traffic.

3.1 Key Technology Decisions

- **Compute (FaaS)**: **AWS Lambda** is utilized for all backend logic. This encompasses API request handling, background image processing (PictureService), and scheduled scheduled tasks (VideoService). This removes the burden of server management (EC2) and aligns costs with actual usage.
- **Storage**:
  - **Amazon S3**: Serves as the backbone for unstructured data storage (Raw Images, Processed Artifacts, Videos). Its event notification system is pivotal for triggering the PictureService.
  - **Amazon RDS (PostgreSQL)**: Chosen for transactional integrity and structured query requirements (e.g., complex filtering by date and city).
  - **Redis (Cloud)**: Implemented as a caching layer (Cache-Aside pattern) to satisfy User Stories 20-21. It stores frequently accessed datasets (like city lists for auto-complete), drastically reducing read latency and database load.
- **API Management**: **AWS API Gateway (HTTP API)** provides a unified entry point, handling routing and acting as a secure facade for the Lambda functions.
- **Infrastructure as Code (IaC)**: **Terraform** is employed to provision all resources, ensuring the infrastructure is reproducible, version-controlled, and transparent.

4 Building Block View

4.1 Level 1: Whitebox Overall System

The high-level architecture is divided into three logical units: the external data producers (Webcams), the central processing core (Weather Archive), and the consumers (Users).

Webcam Authenticates via API Key and pushes raw image data to the system.

Weather Archive System The core serverless boundary encapsulating business logic, storage, and database management.

User Consumes processed content via a Single Page Application (SPA).

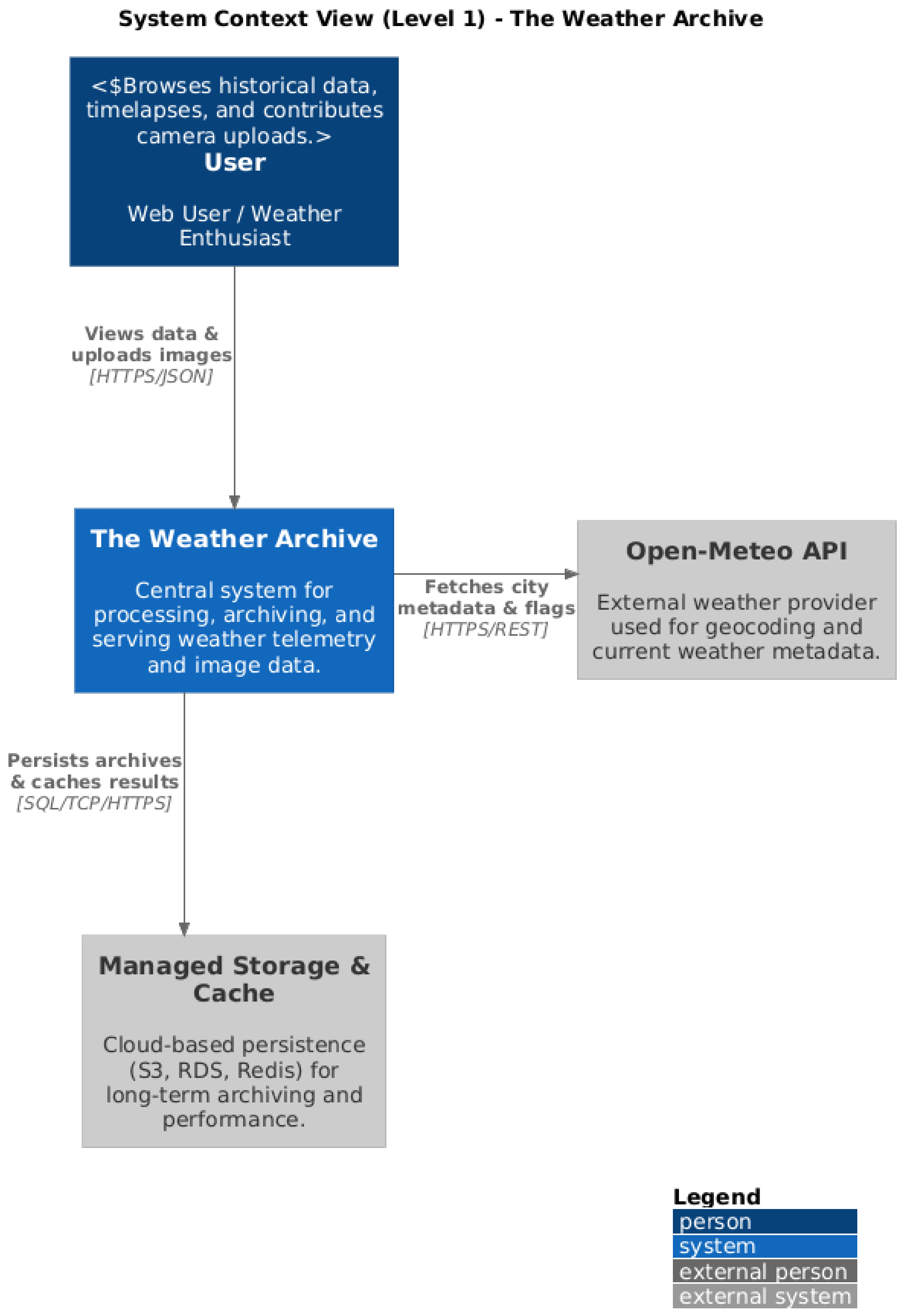
System Context View (Level 1) - The Weather Archive
<$Browses historical data, timelapses, and contributes camera uploads.>
User
Web User / Weather Enthusiast
Views data & uploads images
[HTTPS/JSON]
The Weather Archive
Central system for processing, archiving, and serving weather telemetry and image data.
Fetches city metadata & flags
[HTTPS/REST]
Open-Meteo API
External weather provider used for geocoding and current weather metadata.
Persists archives & caches results
[SQL/TCP/HTTPS]
Managed Storage & Cache
Cloud-based persistence (S3, RDS, Redis) for long-term archiving and performance.
Legend
person
system
external person
external system

Figure 2: Whitebox Overall System (Level 1)

4.2 Level 2: Container View / AWS Resources

The logical containers map directly to managed AWS services, as defined in the Terraform configuration.

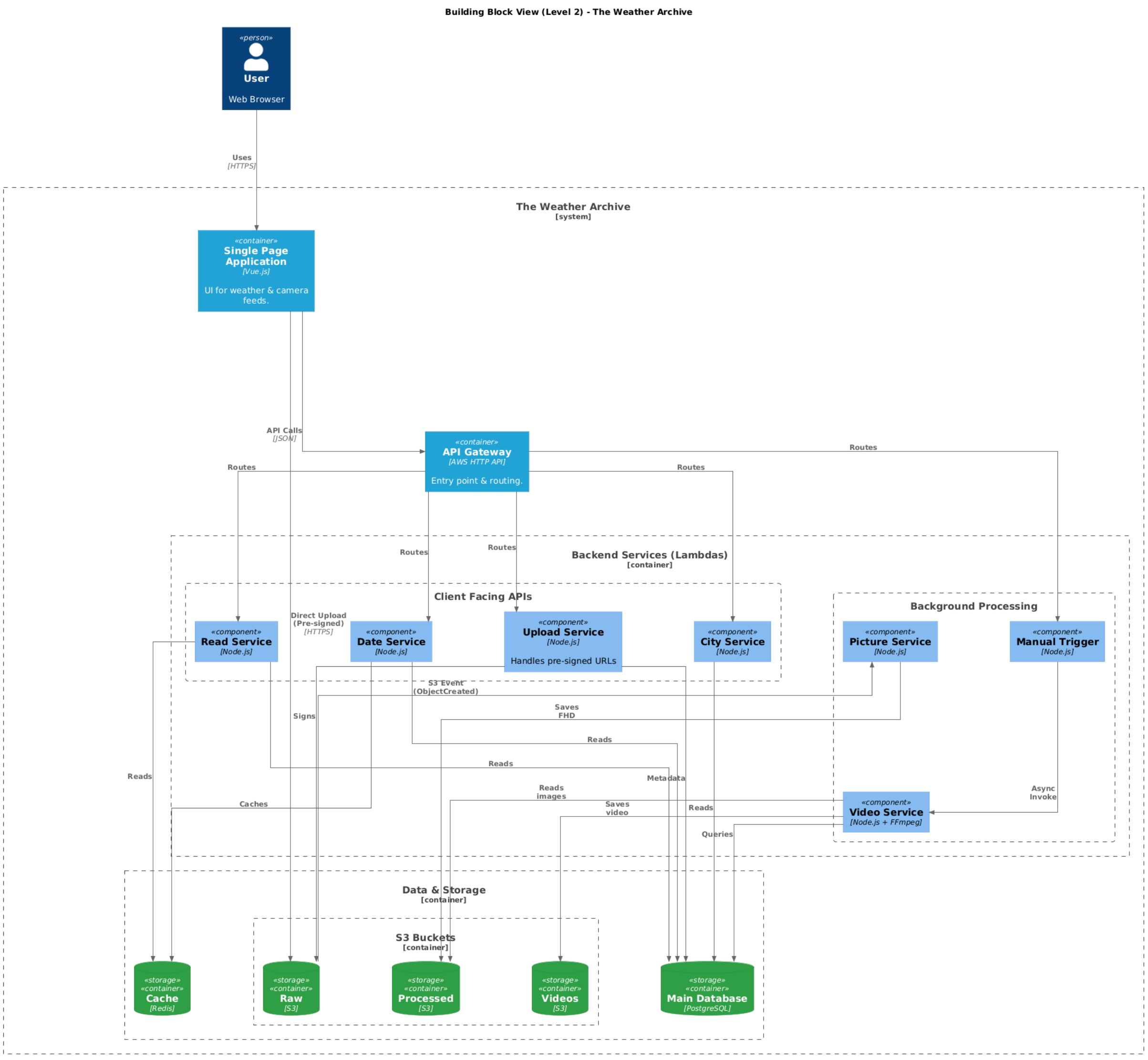

Figure 3: Container View (Level 2)

4.2.1 API Layer (AWS API Gateway)

Resource: `aws_apigatewayv2_api.weather_api`

- Acts as the secure front door for all backend requests.
- **Endpoints**:
  - `GET /upload-url`: routed to `upload_service` to secure upload permissions.
  - `GET /cities`: routed to `city_service` (Cached).
  - `GET /dates`: routed to `date_service` (Cached).
  - `GET /data`: routed to `read_service`.
  - `POST /video/trigger`: routed to `test_trigger_service` for manual invocation.

4.2.2 Compute Layer (AWS Lambda)

The application logic is distributed across specialized micro-functions:

- **UploadService** (`weather-archive-upload`): validates the `x-api-key` header and generates presigned S3 URLs, allowing webcams to upload directly to S3 without passing file data through the Lambda layer (optimizing throughput).
- **PictureService** (`weather-archive-picture`): A purely event-driven function triggered by S3 `ObjectCreated` events. It handles image compression and updates the PostgreSQL metadata.
- **VideoService** (`weather-archive-video`): Invoked by an Amazon EventBridge Schedule (`rate(24 hours)`). It utilizes an FFMPEG Lambda Layer to stitch daily images into time-lapse MP4 videos.
- **Read Services** (`CityService`, `DateService`): These functions implement logic to check Redis before querying RDS, ensuring fast response times for the frontend's autocomplete and calendar features.

4.2.3 Data Layer

- **Amazon S3 Buckets**:
  - `weather-archive-raw-*` (Ingest Quarantine)
  - `weather-archive-processed-*` (Public Read Access via CloudFront)
  - `weather-archive-videos-*` (Generated Content)

- **Amazon RDS**: A PostgreSQL instance (`db.t3.micro`) storing the relational model of Cities, Dates, and Image references. It resides in a private subnet for security.
- **Redis**: An external Redis Cloud instance. It stores transient data such as the specific list of active cities. A TTL (Time-To-Live) strategy allows the cache to refresh automatically, ensuring data consistency with the primary database.

4.3 Level 3: Component View

The Video Service represents the most complex computational component. Its internal workflow allows for efficient media processing within the Lambda time constraints.

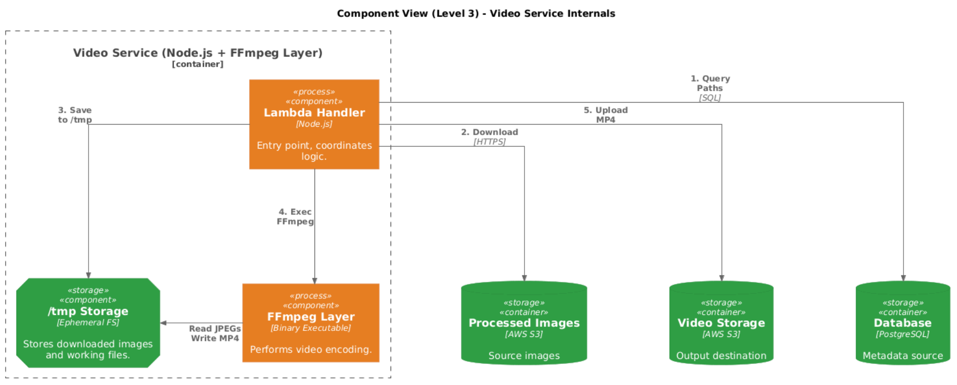

Figure 4: Component View: Video Service (Level 3)

5 Runtime View

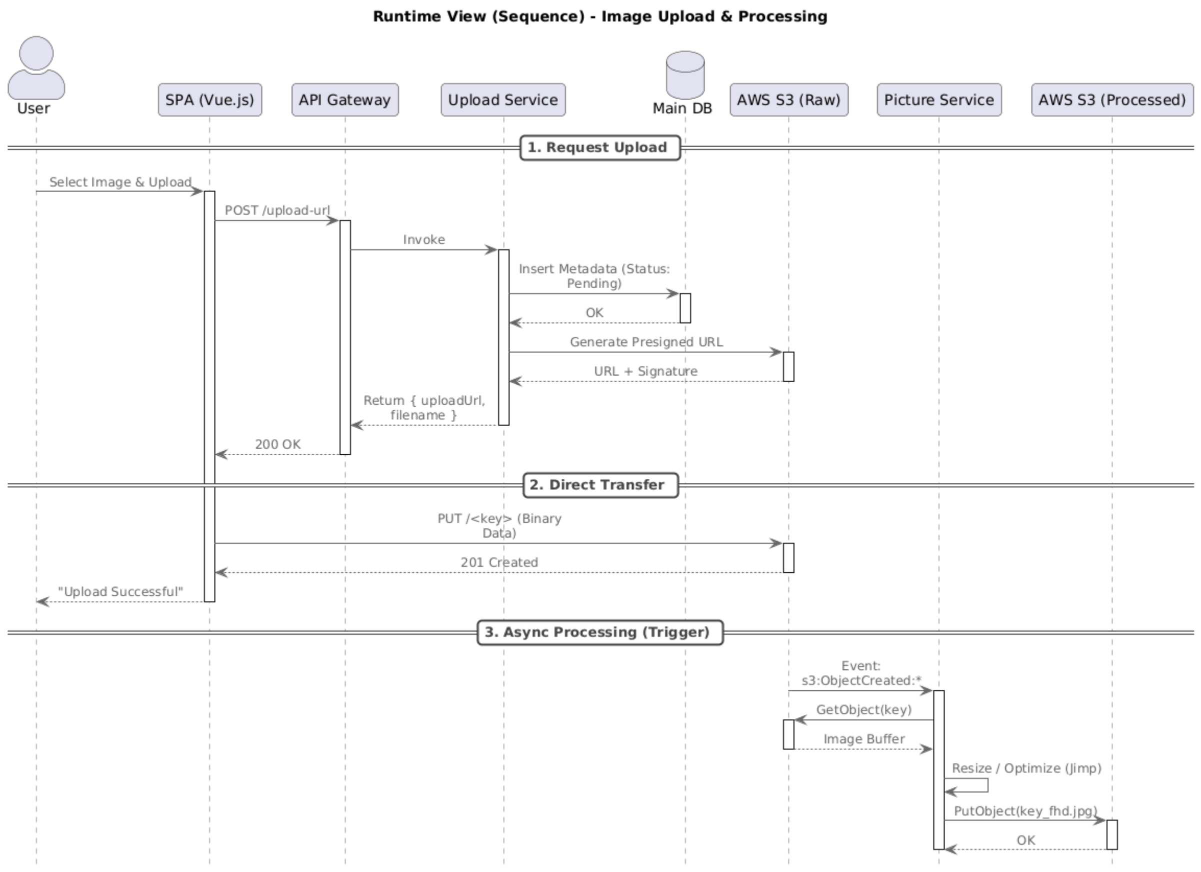

Figure 5: Runtime View: Image Upload Sequence

6 Deployment View

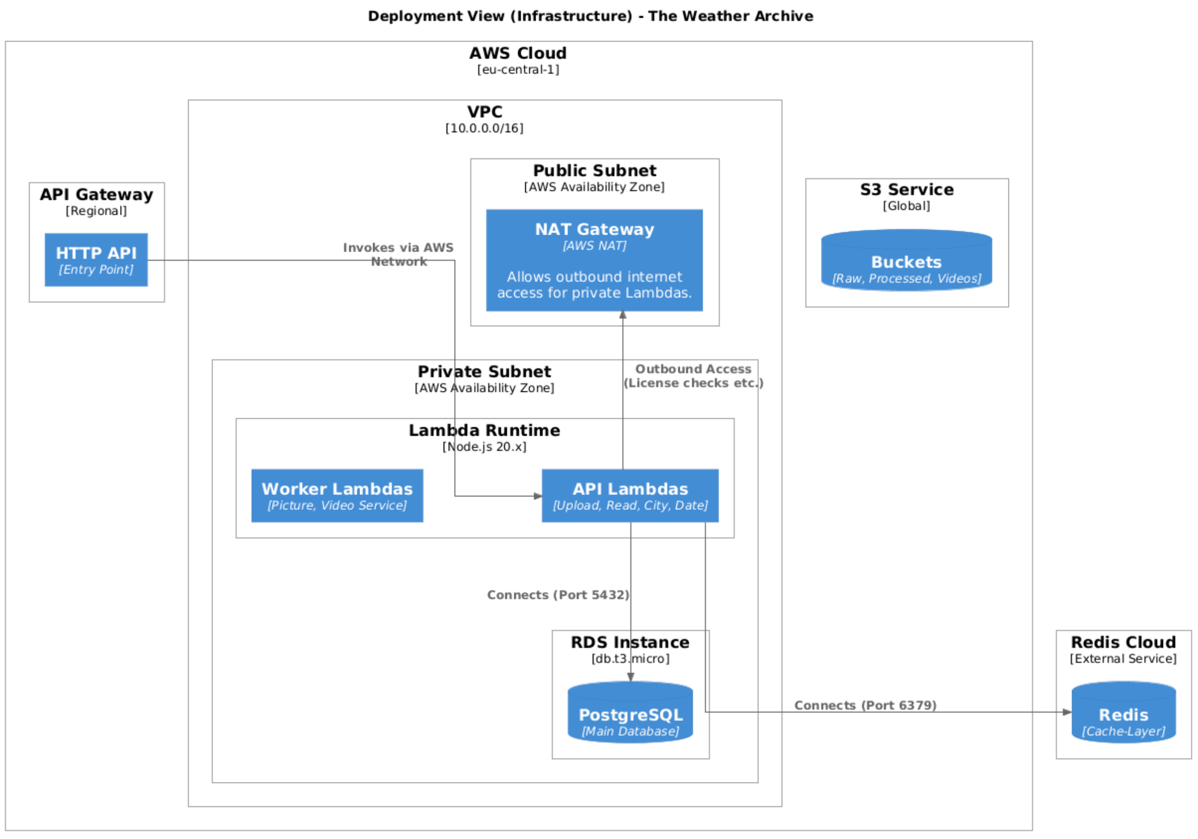

Figure 6: Deployment View: Infrastructure

7 Architecture Decisions

This section records the architectural decision records (ADRs) that shaped the system.

7.1 ADR-1: Adoption of Serverless Compute (AWS Lambda)

Context: The project requires a backend capable of handling two distinct traffic patterns: continuous, sporadic uploads from distributed webcams and bursty read traffic from web users. The system must be cost-effective and low-maintenance.

Decision: AWS Lambda is selected as the primary compute platform.
Justification:

- **Event-Driven Scaling**: Lambda natively handles the asynchronous nature of the Picture-Service (triggered by S3 uploads) without the need for constant polling or long-running worker servers.
- **Cost Optimization**: Given the definition of the "Weather Archive," the system may have periods of inactivity. Lambda's billing model (per 100ms) ensures near-zero cost during idle times, unlike EC2 instances.
- **Operational Simplicity**: Removes the requirement for OS patching and server provisioning, allowing focus on application logic.

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

Context: The specific requirements for filtering weather data by city, date, and time range imply a structured data model with defined relationships.
Decision: Amazon RDS (PostgreSQL).
Justification: SQL provides powerful querying capabilities essential for the search and filtering features. Constraints and schemas ensure data integrity for metadata.

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

Context: The application features high-traffic read endpoints (e.g., retrieving the list of available cities for the search bar) where data changes infrequently.
Decision: Implement Redis with Cache-Aside pattern.
Justification:

- **Latency Reduction**: In-memory retrieval creates a snappy user experience for the "Autocomplete" user stories (US 20-21).
- **Database Protection**: Offloads repetitive read queries from the main RDS instance, preventing bottlenecks during traffic spikes.

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

Context: Image processing is computationally intensive and time-consuming. Performing this synchronously during the webcam upload request would risk timeouts and reduce client reliability.
Decision: Decouple upload and processing using S3 Event Notifications.
Justification: This pattern ensures the webcam receives a fast "200 OK" response immediately after upload. Processing happens reliably in the background, scaling independently of the upload throughput.

7.5 ADR-5: Lambda Concurrency Limiting

Context: The project is hosted on an AWS Academy educational account, which enforces strict resource limits. Exceeding the global concurrent execution limit previously resulted in account deactivation.
Decision: Explicitly configure `reserved_concurrent_executions` for every Lambda function via Terraform.
Justification:

- **Compliance**: The total reserved concurrency across all functions is capped at 9 (below the critical threshold of 10) to prevent account suspension.
- **Resource Management**: Prevents any single function (e.g., a burst of uploads triggering PictureService) from starving other critical functions or exceeding account quotas.