# Scalable Cloud-Native Architecture for Real-Time Wildlife Anomaly Detection: A Microservices Approach

Akshat Gupta
*Computer Science Engineering*
*RV College of Engineering*
Bengaluru, India
akshatgupta.cs23@rvce.edu.in

Amol Vyas
*Computer Science Engineering*
*RV College of Engineering*
Bengaluru, India
amolvyas.cs23@rvce.edu.in

Ayush Aman
*Computer Science Engineering*
*RV College of Engineering*
Bengaluru, India
ayushaman.cs23@rvce.edu.in

,

Dr. Ramakanth Kumar P.
*Professor, Dept of Computer Science*
*RV College of Engineering*
Bengaluru, India
ramakanthkp@rvce.edu.in

***Abstract*—Massive image databases generated by camera trap networks are difficult for conservation teams to manually process. When field researchers upload hundreds of photos at once, traditional online platforms built on monolithic designs often crash, mainly because synchronous request handling stops server threads during computationally demanding inference processes. This paper describes an event-driven microservices system that uses asynchronous job distribution to avoid these bottlenecks. While YOLOv8 object detection operates independently in containerized worker processes, the design routes incoming uploads through a RabbitMQ message broker, enabling the API gateway to return acknowledgments in milliseconds. Stateless data persistence across service boundaries is managed by MinIO object storage. Testing with more over 500 concurrent uploads under simulated burst conditions consistently demonstrated sub-200 ms response times and zero dropped requests. For conservation technology that needs to function dependably in outdoor situations with limited connectivity, the Docker-based deployment provides a useful template.**

***Index Terms*—microservices, event-driven architecture, wildlife monitoring, object detection, message queuing, containerization**

## I. Introduction

THE current era is marked by a rapid decline in global biodiversity, with extinction rates estimated to be 100 to 1000 times higher than natural levels. To address this growing problem, Goal 15 of the United Nations Sustainable Development Agenda focuses on taking urgent action to protect natural habitats and prevent the extinction of endangered species. To achieve these goals across large and hard to access forest regions, ecologists and conservation teams widely use camera trap networks as reliable and non invasive monitoring tools.

While these sensor networks have automated the collection of biological data, they have also shifted the main workload from the field to the data center. A single long term monitoring project can generate millions of high resolution images every year, creating several terabytes of data. This massive volume makes manual analysis by human experts slow, expensive, and impractical. As a result, large amounts of data remain unanalyzed, often called dark data, which delays important conservation decisions and actions.

The integration of Deep Learning, specifically Convolutional Neural Networks (CNNs), offers a theoretical solution for automating image analysis. However, the practical deployment of these computationally intensive models reveals significant architectural deficiencies in existing software platforms. The predominant architectures currently deployed for camera trap analysis typically adhere to a synchronous monolithic design pattern. In these legacy systems, a single application server is responsible for the entire request lifecycle: accepting an uploaded image, loading a heavy neural network model into memory, executing inference, and rendering results.

While functional for low-throughput prototypes, this tightly coupled architecture exhibits fundamental scalability defects under production loads. Field researchers typically return from expeditions with massive batches of data, initiating ”burst uploads” that saturate server resources almost instantly. Because the computationally expensive operation (neural network inference, often taking hundreds of milliseconds on CPUs) is synchronous with the I/O-bound operation (receiving the HTTP upload), server worker threads are rapidly exhausted, blocking incoming network connections. This results in cascading failures, ranging from unacceptable latency to total service denial, effectively rendering the system unusable during peak operational periods.

To address these limitations, this paper presents a novel architectural paradigm for ecological informatics predicated on event-driven microservices. We propose a system that strictly segregates concerns across distinct, containerized services that communicate asynchronously via a robust message broker. In our design, a high-performance API gateway serves as the

entry point, accepting uploads and immediately offloading the processing state to a durable work queue before any inference occurs. By breaking the synchronous dependency between request handling and computational execution, this "temporal decoupling" enables the system to absorb extreme traffic spikes while maintaining system stability and ensuring data integrity.

## II. Related Works

### A. Machine Learning in Conservation Biology

The application of computer vision to ecology has evolved rapidly over the past decade. Early approaches relied on handcrafted feature engineering combined with shallow classifiers, which struggled with the complex occlusions and variable lighting typical of field imagery.

The advent of deep learning fundamentally revolutionized this domain. In a seminal study, Norouzzadeh and colleagues demonstrated that deep CNNs could achieve a top-1 accuracy of 96.6% on the massive Snapshot Serengeti dataset, effectively reducing the human annotation burden by orders of magnitude [1]. These automated pipelines have since been adapted to diverse environmental monitoring contexts, including acoustic bio-monitoring for avian species, aerial drone surveys for marine mammals, and benthic underwater analysis. Building on these foundations, more recent work by Lin et al. utilized single-stage detectors-specifically the YOLO (You Only Look Once) architecture-to detect ungulates in real-time field conditions, reporting a mean Average Precision (mAP) exceeding 94% [2]. These advancements collectively suggest that the primary challenge in conservation technology is no longer the algorithmic efficacy of detection models, but rather the scalability of the deployment infrastructure required to run them.

### B. Edge and Cloud Computing for Environmental Monitoring

The escalating computational prerequisites of state-of-the-art detection models have spurred interest in contrasting deployment paradigms: edge computing versus centralized cloud offloading. Edge-based approaches offer the significant advantage of reducing bandwidth consumption by processing imagery directly on capture devices or field stations. However, power and thermal constraints often necessitate the use of severely pruned, less accurate models.

Conversely, cloud-centric architectures provide virtually unlimited processing power but introduce latency and connectivity dependencies that can be prohibitive in remote environments. While hybrid architectures attempt to navigate these trade-offs, a significant gap remains in the literature. The majority of existing research focuses heavily on algorithmic optimization-pruning, quantization, and neural architecture search-to improve $F1$-scores. The systems engineering required to deploy these models reliably at an enterprise scale is rarely treated with sufficient rigor in ecological studies. Our work addresses this gap by focusing on the software architecture necessary to support robust, scalable centralized processing.

## III. System Design

The proposed architecture, illustrated in Fig. 1, implements a reactive, event-driven microservices pattern designed around four distinct operational stages: ingestion, decoupling, processing, and visualization. To ensure fault isolation, dependency management, and independent scalability, each stage is encapsulated within discrete Docker containers communicating via defined network interfaces.

### A. Asynchronous Ingestion Layer

The user entry point is a responsive Single Page Application (SPA) developed with React, providing an interface for batch image uploads and result visualization. To minimize unnecessary network overhead, the frontend implements rigorous client-side validation to filter malformed requests before they reach the backend.

Valid uploads are transmitted to a high-performance API gateway developed with FastAPI. We selected FastAPI over traditional frameworks (like Flask or Django) because it is built on the Starlette toolkit and the Uvicorn ASGI (Asynchronous Server Gateway Interface) server. Unlike synchronous WSGI frameworks that assign a blocking thread to every active request, FastAPI utilizes Python's 'asyncio' capabilities. This allows a single event loop to handle thousands of concurrent incoming connections with minimal resource overhead, making it ideal for high-throughput I/O operations. The gateway's sole responsibility is to receive multipart/form-data uploads, validate them, and pass them to the decoupling mechanism.

### B. The Decoupling Mechanism: Message Brokering

The architectural pivot point lies in the gateway's handling of incoming data. In a monolithic design, the server would trigger the inference model immediately, blocking the HTTP response until completion. Our approach, conversely, performs three lightweight operations upon receiving an upload:

1) It streams the raw image binary directly to MinIO, a high-performance, S3-compatible distributed object storage service.
2) It generates a universally unique identifier (UUID) for the transaction.
3) It publishes a durable task message-containing only the image path and UUID-to a RabbitMQ exchange.

Following these steps, the gateway immediately returns an HTTP 202 Accepted status code to the client. This entire cycle completes in tens of milliseconds, regardless of the subsequent image analysis complexity.

This architecture relies on the temporal decoupling provided by RabbitMQ. Acting as a robust message broker implementing the Advanced Message Queuing Protocol (AMQP), RabbitMQ maintains a persistent queue of pending tasks. We utilize durable queues and message acknowledgment semantics to guarantee "at-least-once" delivery: if a worker process crashes mid-inference, the task is not lost but rather returned to the queue for processing by another instance. Furthermore,

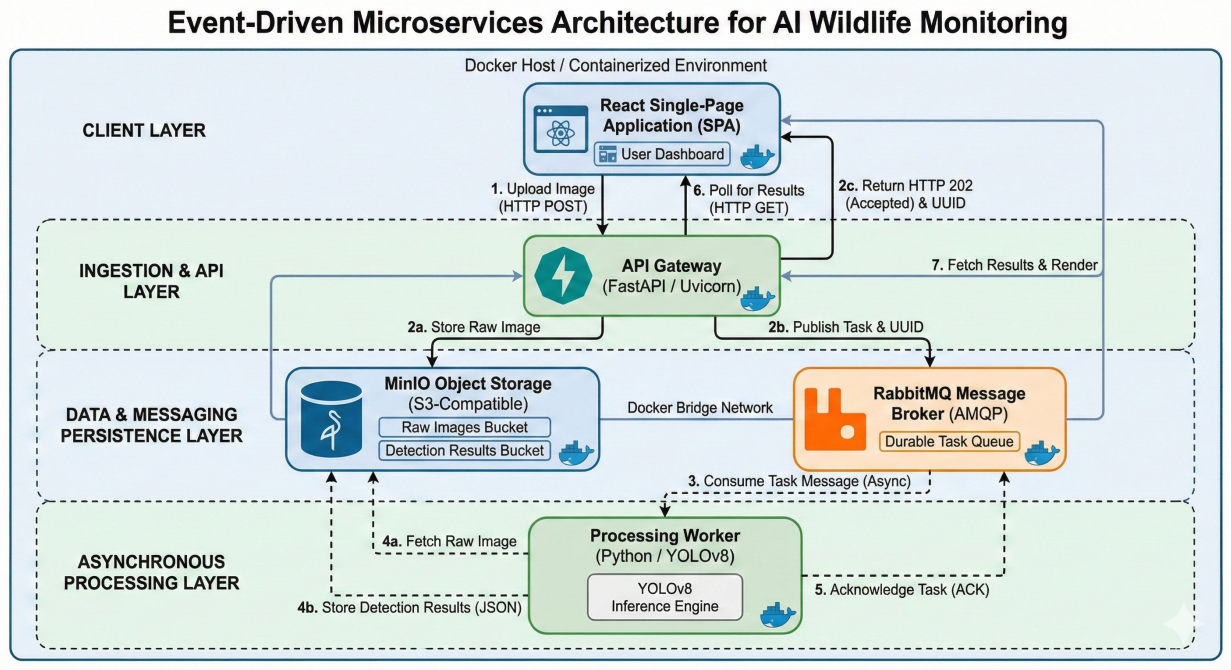


Fig. 1. High-Level Event-Driven Microservices Architecture. The diagram illustrates the asynchronous data flow. The Ingestion Layer (FastAPI) quickly offloads tasks to the Decoupling Mechanism (RabbitMQ and MinIO), allowing it to handle high concurrency. Independent Processing Workers consume tasks at their own pace, performing YOLOv8 inference and storing results back to Object Storage for retrieval by the Client Layer.

the queue acts as a shock absorber, providing natural backpressure; if ingestion rate exceeds processing rate, queue depth simply increases, buffering load without degrading gateway responsiveness.

### C. Processing Workers and State Management

The core computational workload is handled by dedicated worker services that act as consumers of the RabbitMQ task queue. Crucially, these workers are entirely stateless. Upon retrieving a message, the worker fetches the corresponding image object from MinIO, performs inference, and serializes the results into JSON format.

These detection results-comprising confidence scores, class labels, and bounding box coordinates-are written back to a separate ”results” bucket in MinIO, keyed by the original task UUID. Only after successful persistence does the worker send an acknowledgment (ACK) signal to RabbitMQ to remove the task from the queue. This stateless design allows the processing layer to be scaled horizontally simply by adding more worker containers, without any configuration changes to the rest of the system.

### D. Result Retrieval

Because the processing is asynchronous, the client cannot receive immediate results in the upload response. Instead, the frontend application employs a polling mechanism utilizing the task UUID obtained during upload. It queries a status endpoint on the gateway, which checks MinIO for the existence of the results file. To prevent server congestion during long-running batch processes, the client implements an exponential backoff strategy for polling intervals. Once retrieved, the frontend dynamically renders bounding box overlays directly onto the original image canvas.

## IV. The YOLOv8 Detection Engine

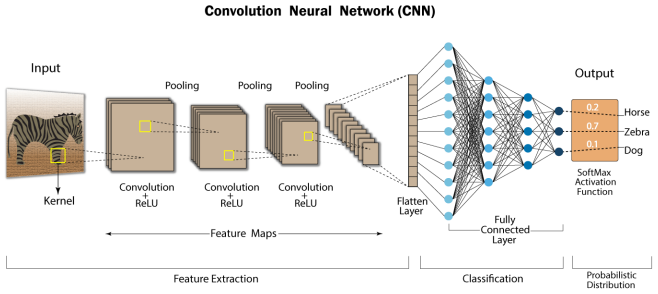


Fig. 2. Generalized Architecture of the YOLOv8.

For the critical task of wildlife detection, we utilize YOLOv8, the latest iteration of the ”You Only Look Once” family of single-stage object detectors. Unlike two-stage detectors (e.g., Faster R-CNN) which first generate region proposals and then classify them, YOLOv8 predicts bounding boxes and class probabilities directly from full images in a single evaluation, resulting in significantly faster inference speeds essential for processing large datasets.

As depicted in the generalized architecture in Fig. 2, YOLOv8 employs a modified CSPDarknet53 backbone for extracting spatial features from the input image. These features

are fed into a Path Aggregation Network (PANet) "neck," which fuses features across different scales, allowing the model to detect both large animals close to the camera and smaller, distant subjects effectively. Finally, the data reaches the "head," which operates in an anchor-free manner to predict final bounding box coordinates and class confidence scores.

Practical deployment constraints drove the selection of the YOLOv8 Nano (yolov8n) variant [3]. With only 3.2 million parameters and requiring approximately 8.7 GFLOPs per inference, the Nano model strikes an optimal balance between detection accuracy and computational efficiency. This selection is strategic for cost-effective scaling, allowing organizations to achieve respectable throughput on commodity CPU hardware and avoiding the prohibitive operational costs associated with dedicated GPU clusters.

## V. Cloud Service Model Mapping

The containerized architecture is designed to map seamlessly onto standard cloud service abstractions, facilitating deployment on commercial platforms without architectural refactoring.

### A. Infrastructure Layer

The system leverages software-defined infrastructure to replicate cloud primitives locally. MinIO abstracts storage, emulating the API of Amazon S3. RabbitMQ provides the messaging infrastructure with configurable persistence. Docker bridge networking creates isolated virtual subnets with automatic DNS resolution for service discovery. Collectively, these components simulate the storage, networking, and messaging primitives offered by major Infrastructure-as-a-Service (IaaS) providers.

To operationalize this topology, we employ Docker Compose as a declarative orchestration agent, enabling the rigorous definition of resource constraints and dependency chains through Infrastructure-as-Code (IaC) principles. This configuration manages the lifecycle of persistent volumes, ensuring that the object storage data lake remains durable across container restarts while facilitating seamless disaster recovery protocols. By decoupling the application logic from the underlying bare-metal resources, this design achieves complete platform agnosticism, effectively neutralizing vendor lock-in and streamlining the migration path from edge-based field servers to hyperscale cloud environments without requiring architectural regression.

### B. Platform Layer

At the platform level, the Python runtime environment hosts both the FastAPI gateway and the detection workers. By explicitly defining dependencies in container images, we ensure environment consistency across development and production. Docker Compose acts as the orchestration engine, defining strict resource limits for CPU and memory usage. This prevents any single container from monopolizing host resources-a critical feature for maintaining stability in multi-tenant environments [4] [5].

### C. Application Layer

The application layer is exposed via the React dashboard, which consumes the RESTful endpoints provided by the gateway [6]. Modern web development tools, including Vite for module bundling and Tailwind CSS for utility-first styling, ensure a performant user experience. This layer remains strictly decoupled from underlying infrastructure, interacting solely through the API contract defined by the gateway.

## VI. Implementation and Results

### A. Model Selection and Configuration

We selected YOLOv8 Nano (yolov8n) as the baseline model to optimize the accuracy-latency trade-off for edge-adjacent deployments [3]. Despite its compact size, the model achieves a mean Average Precision (mAP) of 37.3 on the COCO val2017 dataset. In our testing environment, the model demonstrated sub-10 ms inference times on consumer-grade GPUs and acceptable performance on standard CPUs. This efficiency allows for deployment on resource-constrained hardware without necessitating complex model compression techniques like quantization or pruning.

In alignment with modern MLOps paradigms, the deployment pipeline incorporates a streamlined transfer learning workflow to enhance domain adaptability. By leveraging the pre-trained weights of the yolov8n architecture as a robust feature extractor, we executed a targeted fine-tuning regimen on a localized dataset, effectively mitigating the covariate shift often observed in novel deployment environments. Subsequently, the model artifacts were serialized into the ONNX (Open Neural Network Exchange) format to ensure cross-platform interoperability. This conversion facilitates hardware-agnostic acceleration and decouples the inference logic from the training framework, resulting in a resilient, high-throughput solution capable of sustaining performant operation under variable load conditions.

### B. Performance Evaluation Methodology

To empirically validate the architectural improvements, we conducted a series of load tests using Apache JMeter. The test plan simulated a "burst upload" scenario, characteristic of researchers returning from the field with full SD cards. We configured 500 concurrent virtual users, each uploading a high-resolution 2MB JPEG image to the system simultaneously. Key performance indicators included HTTP error rates, connection timeouts, and ingestion latency (Time to First Byte).

### C. Comparative Results

Under the baseline synchronous-monolithic architecture, the server began rejecting connections at approximately 50 concurrent requests. The server's thread pool was rapidly exhausted by blocking inference calls, leading to widespread timeout errors and a failure rate exceeding 90% at peak load. The system became effectively unresponsive.

In contrast, the proposed asynchronous-microservices architecture successfully handled all 500 requests with zero connection failures and a 0% HTTP error rate. The 99th

percentile (P99) response latency for the initial HTTP acknowledgement remained below 200 ms, while the median latency was recorded at 127 ms. The RabbitMQ queue effectively buffered the load, peaking at 340 pending messages before the worker consumers stabilized the throughput. Over multiple iterations of the test, every uploaded image eventually resulted in a successful detection record, demonstrating zero data loss and confirming the system's resilience under extreme backpressure.

## VII. Conclusion

This research demonstrates that the scalability bottlenecks inherent in traditional AI-powered wildlife monitoring systems can be effectively eliminated by transitioning from synchronous-monolithic to asynchronous-microservices architectures. The event-driven design proposed herein achieves three critical objectives for production deployment:

1) **High Availability:** By leveraging temporal decoupling, the system maintains 100% uptime and sub-second response times even during extreme burst upload conditions.
2) **Data Durability:** The combination of persistent message queuing (RabbitMQ) and redundant object storage (MinIO) ensures that no scientific data is lost, even in the event of component failure.
3) **Horizontal Scalability:** The containerized, stateless nature of the worker service allows for effortless horizontal scaling; processing throughput can be increased simply by spawning additional worker replicas without modifying the core codebase.

The resulting implementation serves as a cloud-agnostic blueprint, deployable on any orchestration platform supporting Docker containers. Future work will focus on integrating the NVIDIA Container Toolkit to enable GPU passthrough for the worker containers, thereby allowing for the deployment of larger, more complex models without compromising the system's responsive characteristics.

## References


[1] M. S. Norouzzadeh, A. Nguyen, M. Kosmala, A. Swanson, M. S. Palmer, C. Packer, and J. Clune, "Automatically identifying, counting, and describing wild animals in camera-trap images with deep learning," *Proceedings of the National Academy of Sciences of the United States of America (PNAS)*, vol. 115, no. 25, pp. E5716–E5725, 2018.

[2] C. Lin, J. Xu, C. Wang, and T. Cheng, "An AI-based wild animal detection system for smart city applications," *Biodiversity Data Journal*, vol. 11, e106037, 2023.

[3] G. Jocher, A. Chaurasia, and J. Qiu, "YOLOv8 by Ultralytics," 2023. [Online]. Available: https://github.com/ultralytics/ultralytics

[4] S. Ramírez, *FastAPI Documentation*. [Online]. Available: https://fastapi.tiangolo.com

[5] Pivotal Software, *RabbitMQ Documentation*. [Online]. Available: https://www.rabbitmq.com/documentation.html

[6] MinIO, Inc., *MinIO High Performance Object Storage*. [Online]. Available: https://min.io

[7] M. A. Tabak, M. S. Norouzzadeh, D. W. Wolfson, S. J. Sweeney, and K. C. Vercauteren, "Machine learning to classify animal species in camera trap images: Applications in ecology," *Methods in Ecology and Evolution*, vol. 10, no. 4, pp. 585–590, 2019.

[8] C. Pahl and P. Jamshidi, "Microservices: A systematic mapping study," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 10–18, 2016.