ReturnZero_

28 September 2025
COS301 - Capstone
University of Pretoria

BushBuddy

BushBuddy Requirements Specification

Version 4

Team members
Ruan Esterhuizen (u23532387)
Ruben Hannes Gadd (u23633353)
Raphael Rato (u22887581)
Tom Schulz (u05039364)
Jean Steyn (u22537229)

Contact
g24capstone@gmail.com

Contents

Introduction

Our vision is an intuitive application for users to reliably identify African wild animals using an AI-powered image detection system

Our mission is for our platform to promote awareness, education and appreciation for South African wildlife. We seek to cultivate a community of users who are inspired to explore local nature reserves and contribute to conservation efforts

Goals

1. **Accurate animal identification**
   Utilize an AI-powered image detection system that accurately identifies African wildlife species based on image and/or audio input.

2. **Location-Based Insights**
   Users can explore sightings by using an interactive map that allows real-time discovery and exploration of wildlife sightings.

3. **Community and social engagement**
   Provide a social platform where users can share sightings with friends and interact with others.

4. **Gamification and exploration**
   Encourage engagement and provide an educational platform using a gamified achievements and bestiary system.

User Characteristics

Users of the BushBuddy application falls into 2 categories:

1. Normal users (general public)
2. Admins (Trusted individuals, responsible for content moderation)

A normal user can further be defined as anybody who has a smartphone that meets the minimum requirements for the BushBuddy application and is interested in visiting nature reserves.

User Stories

1. User Stories: User

1.1 AI Detection

#	User Story	Acceptance Criteria
1	As a user, I want to upload an image so that I can identify what animal I see.	- Prompt user for photo album access - User can upload image - App should identify what the animal
2	As a user, I want to take photos by using the app, so that I don't need to upload the images	- Prompt user for camera access - User can use their camera from within the app - Model should be able to identify animals with the provided footage
3	As a user, I want to use my camera so that I can identify animals I see in real time.	- Prompt user for camera access - User can use their camera from within the app - Live identification should happen(Identification without images). - Model should be able to identify animals with the provided footage
4	As a user, I want to upload audio so that I can identify an animal by its call.	- Prompt user for audio media access - User is able to upload audio to the app - App is able to identify animal by making use of the sound
5	As a user, I want to use my microphone so that I can identify animal calls in real time.	- Prompt user for microphone access - User is able to use the app for real-time audio collection - The app should be able to collect live audio input from the microphone - The app is able to identify the animal based on the provided input

1.2 Animal Discovery

#	User Story	Acceptance Criteria
1	As a user, I want to open up a map so that I can see where animals have been identified.	- User can open a map in the App -The map should provide pins of sightings - Pin Symbols should be descriptive according to type of animal sighting - Pins should be interactable - Pressing a pin shows details about the sighting
2	As a user, I want to search and filter discoveries so that I only see discoveries that are relevant to me.	- User can search discoveries by keywords (e.g., name, species, location)

		- User can filter results by various criteria (example, animal type, date, proximity, etc)
3	As a user, I want to browse through a lexicon of animals so that I can learn more about them.	- App should have bestiary containing all the apps animals - Each animal in the bestiary is interactable - Tapping an animal icon displays details (e.g., habitat, diet, interesting facts)
4	As a user who has just identified an animal, I want to view detailed information about it, so that I don't have to search for the information elsewhere.	- App should provide information when animals have been identified/discovered (Will be the same information as the Bestiary most likely)
5	As a registered user, I want to see a history of my past discoveries so that I can revisit them at a later time.	- The app stores all discoveries made by the user - Users can view a list or timeline of their previous discoveries

1.3 Social Feed

#	User Story	Acceptance Criteria
1	As a registered user, I want to create posts about identified animals so that I can share my discoveries with other users.	- Registered Users can make posts - Registered Users can allows posts to show the location where the post was taken
2	As a registered user, I want to see other users' posts so that I can connect with people who are in my area.	- Registered Users can view other user posts - Registered Users can filter posts(In this case, based on proximity)
3	As a registered user, I want to add other users as friends so that I can get notified anytime they make a new discovery.	- Registered Users can send/receive and accept/deny friend requests - Registered Users can view sent/received friend requests - Registered User will get notified if their friends make a post/discover an animal
4	As a registered user, I want to be able to see the location where a post was taken.	- Registered Users can view location of posts - Registered users can view locations
5	As a registered user, I want to be able to report posts that seem unsafe, so that animals and their locations can be protected.	- Registered Users can report posts - Users must provide a message/reason with the report - Reports are logged and associated with the reporting user for review

1.4 Account Options

#	User Story	Acceptance Criteria
1	As a registered user, I want to follow specific animals so that I get notified anytime one of them is discovered near me.	- Registered users can follow specific animals - Users receive notifications when a followed animal is discovered near their location
3	As a new user, I want to register with my email or phone number so that I can save my data and discoveries.	- New users can register using an email address or phone number - Upon registration, the app creates an account for saving discoveries and preferences
4	As a user, I want to set my account to public or private so that I can decide whether to share my data/discoveries.	- Users can toggle account visibility between public and private - Private accounts restrict others from viewing discoveries unless granted access

2. User Stories: Admin

2.1 Admin actions

#	User Story	Acceptance Criteria
1	As an admin, I want to access logs from the AI model so that I can make sure it stays accurate and retrain or improve it if necessary.	- Admins can view AI model logs - Admins can view statistics related to the AI model's performance (e.g., confusion matrix, precision, accuracy, recall, etc.)
2	As an admin, I want to access usage logs for uploads and detection so that I can see and understand usage trends.	- Admins can view upload usage logs - Admins can view Detection usage logs - (Optional) Admins can view statistics on recent popular uploads or frequent detections
3	As an admin monitoring reported posts, I want to be able to remove posts that are unsafe, so that the locations of protected animals remain confidential.	- Admins can view a log/list of reported posts - Admins can review reported content - Admins can remove posts after reviewing them

Functional Requirements

FR1: Authentication and Authorisation

- FR1.1: Users should be able to register an account
  - FR1.1.1: Users should be able to register with an email address
  - FR1.1.2: Users should be able to register with their phone number
- FR1.2: Users should be able to log into their account
- FR1.3: Users should be able to log out of their account
- FR1.4: Users should be able to reset their passwords

FR2: Account Management

- FR2.1: Users should manage account settings
  - FR2.1.1: Users should be able to switch their account between private/public
  - FR2.1.2: Users should be able to toggle their Geolocation to On/Off
- FR2.2: Users should be able to edit their credentials such as email, phone number or password
- FR2.3: Users should be able to see users that sent them friend requests
- FR2.4: Friend Request Handling
  - FR2.4.1: Accepting friend requests
  - FR2.4.2: Declining friend requests

FR3: Animal Identification

- FR3.1: Users should be able to identify animals by using their camera
  - FR3.1.1: Identification by uploading an image
  - FR3.1.2: Identification in real-time by using the apps camera interface
- FR3.2: Users should be able to identify animals via audio
  - FR3.2.1: Identification by uploading an audio clip to the app
  - FR3.2.2: Identification in real-time by making use of the user’s microphone
- FR3.3: (Optional)Identification confidence should be shown to the user after prediction
- FR3.4: (Optional) Users can confirm if the identification is correct

FR4: Discovery

- FR4.1: Users should be provided information on animals once identified
- FR4.2: Interactive Geolocation System
  - FR4.2.1: Users should be able to view a map
  - FR4.2.2: The map should have pins of locations where animals were identified
  - FR4.2.3: Pins should be interactable, and display information related to the sighting
  - FR4.2.4: Users should be able to filter the map (by animal type, date, etc)
- FR4.3: Users should be provided a Bestiary
  - FR4.3.1: It should contain all the animals the app is able to identify
  - FR4.3.2: Animal icons should be interactable
    - FR4.3.2.1: All the information should be displayed to the user once interacted with
  - (Optional) FR4.3.3: Users should be able to favourite animals for faster access
  - FR4.3.4: Users should be able to search for animal in the bestiary
  - FR4.3.5: Users should be able to filter animals in the bestiary(by animal type for example)
- FR4.4: Users should be able to follow animals
  - FR4.4.1: By making use of the bestiary after finding the animal
  - FR4.4.2: After successful identification of an animal
- FR4.5: Users have a list/log of past animals they identified with the app

FR5: Social Platform

- FR5.1: Provide Post functionality for identifying animals
  - FR5.1.1: Users should be able to Post images of identified animals
  - FR5.1.2: Protected animals' locations may not be allowed to be shared in the post
  - FR5.1.3: Users should be able to remove their own posts
  - FR5.1.4: Users should be able to report unsafe posts
- FR5.2: Users should be able to interact with other User profiles
  - FR5.2.1: Users should be able to view other user profiles
  - FR5.2.2: User profiles should only display content when set to public
  - FR5.2.3: Users should be able to send Friend Requests to other users

- FR5.3: Users should control what is displayed on their posts
    - FR5.3.1: If Geolocation should be shared
    - FR5.3.2: Adding descriptions to their posts

FR6: User Data Management

- FR6.1: User details should be securely transmitted
- FR6.2: User details should be securely stored
- FR6.3: Users should be prompted for access to their devices for certain features to be enabled
    - FR6.3.1: Prompted for Camera access
    - FR6.3.2: Prompted for Microphone access
    - FR6.3.3: Prompted for Media Album access
    - FR6.3.4: Prompted for Geolocation access

FR7: Admin Functionalities

- FR7.1: Admins should be able to view reported posts
- FR7.2: Admins can delete posts deemed unsafe
- FR7.3: Admins can view AI model performance statistics
- FR7.4: Admins should be able to view usage statistics
    - FR7.4.1: Uploads
    - FR7.4.2: Animal Identifications
- FR7.5: Admins should have normal user functionalities

FR8: AI Model Integration

- FR8.1: The system should log AI model predictions and results
- FR8.2: The system should display performance metrics of the model, like accuracy, precision and other related performance metrics
- FR8.3: Model logs should only be accessible by admins
- (Optional following on FR3.4) FR8.4 System should allow for feedback loops from user validation to improve model accuracy
- (Optional following on FR8.2) FR8.5: AI inference latency should be tracked, and displayed

FR9: Notifications

- FR9.1: Friend requests notifications
- FR9.2: Notifications for animals being followed and spotted near the user
- FR9.3: Notifications if a user's post has been removed or delayed before postings
- FR9.4: Notifications if users received/accepted a friend request
- (Optional)FR9.5: Users should be able to be view a notification log/list

Non-Functional Requirements

Scalability

The system should be able to handle multiple concurrent users and detection requests.

Security

The system should use end-to-end encryption for user data, and also make use of access control in order to make sure users can't access anything they shouldn't be allowed to. The system should also protect endangered animals by not allowing geolocation data to be shared upon detection.

Availability

The application should be able to be used by a wide audience, i.e. it should run smoothly and efficiently on most devices. The application should also make use of user-friendly elements that cater to the needs of many different users (e.g. support for screen readers for the visually impaired, etc.).

Performance

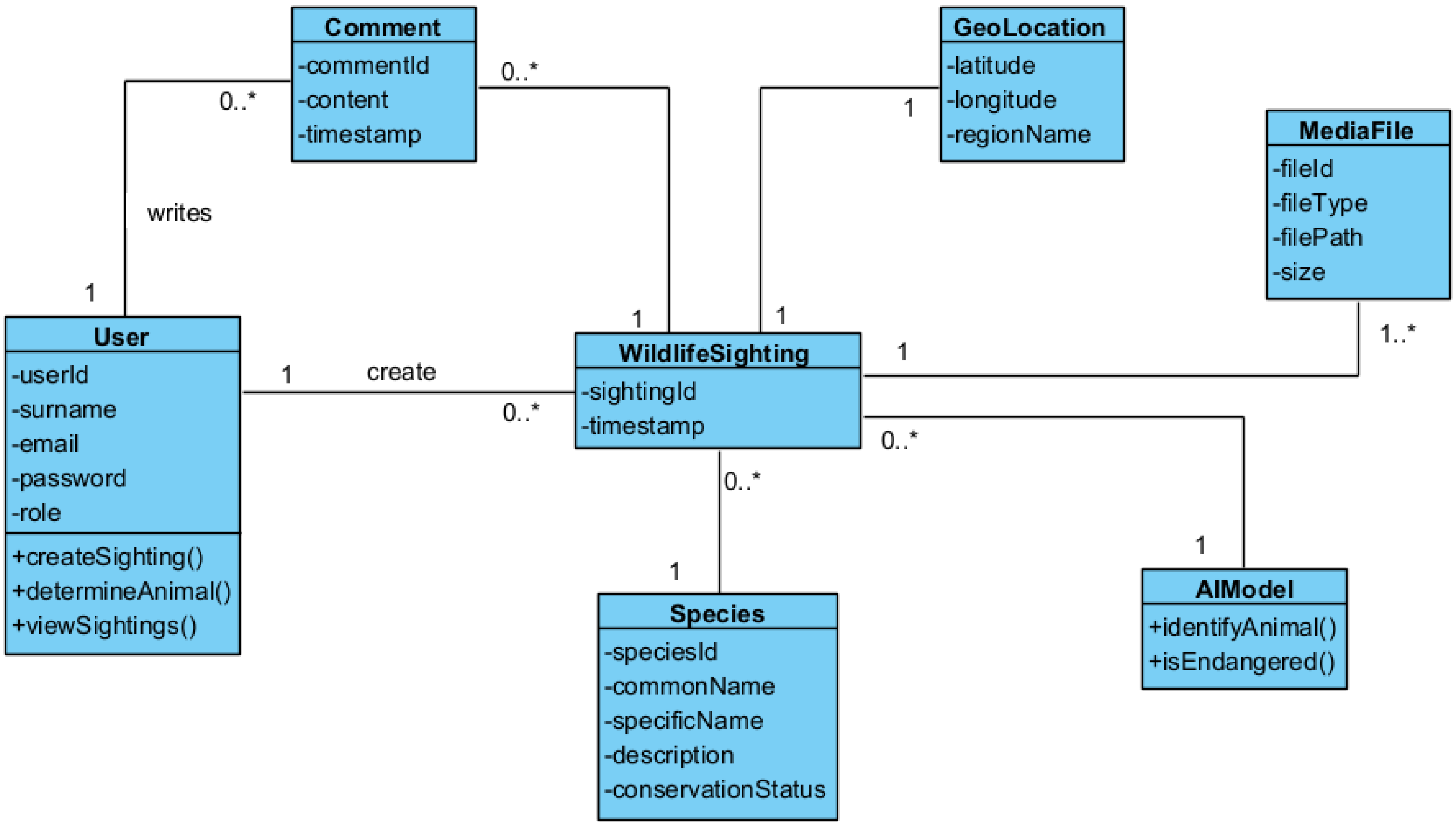
The system should efficiently identify animals through images and/or audio and respond within an acceptable amount of time under typical load. To speed up the identification process, the application will have the option to use the AI locally to detect animals in real-time, which will be a toggleable option to maintain availability.

Reliability

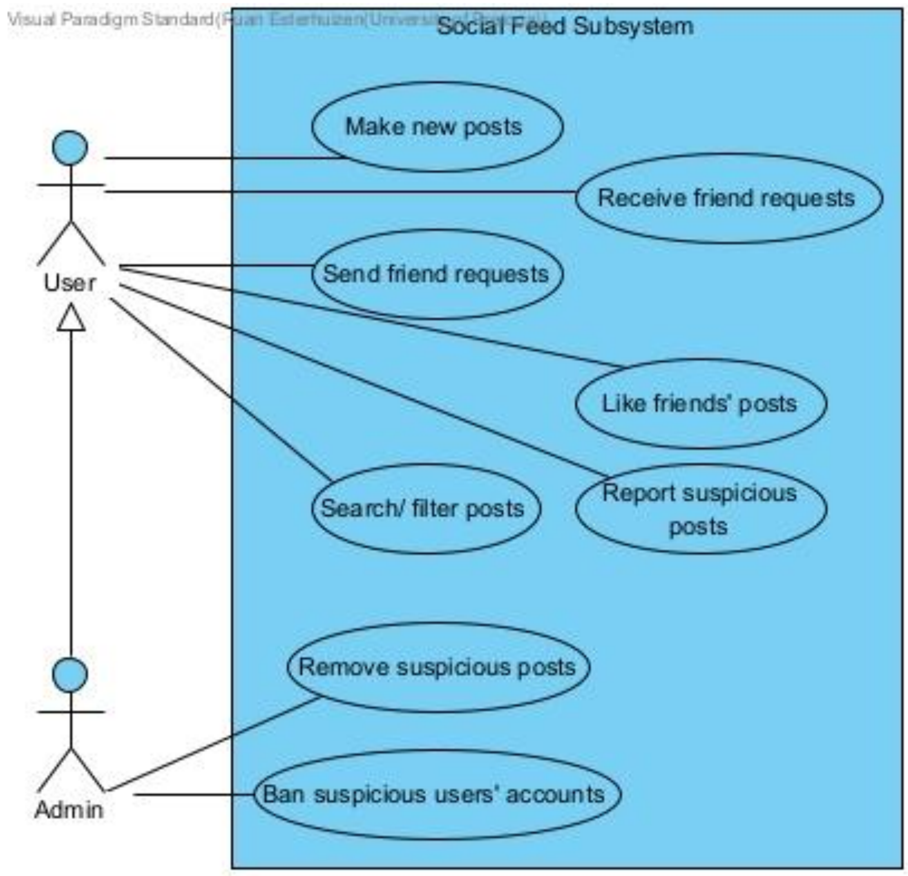
The system should have an accurate identification process to consistently make the correct detections. The system will have fall-back measures for redundancy and added reliability.

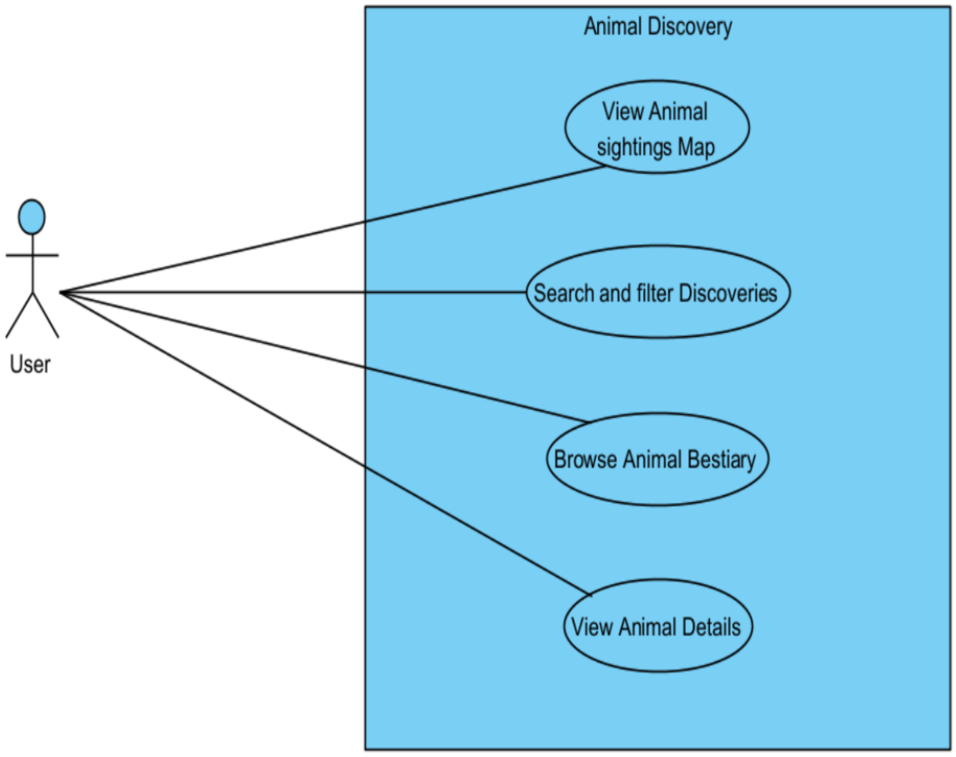
Note: Quality requirements are quantified in the Architectural Requirements

Domain Model



Use-Case Diagrams





Service Contracts

Authentication Service

Login

Endpoint: POST /auth/login

Description:

Authenticates existing user and sets an HTTP-Only cookie with a token.

Request Body

```
{
    "username" : "example user",
    "password" : "password123"
}
```

Response

```
{"message" : "User logged in successfully"}
```

Register

Endpoint: POST /auth/register

Description:

Creates a new user and sets an HTTP-Only cookie with a token.

Request Body

```
{
    "username" : "New user",
    "email" : "newuser@email.com",
    "password" : "password123"
}
```

Response

```
{"message" : "User registered successfully"}
```

Discovery Service

Bestiary

Endpoint: GET /discover/bestiary

Description:

Returns a list of animals that are identifiable by the AI model.

Response

```
{
    "success" : true,
    "message" : "Animals retrieved successfully",
    "data" : [
        {"id": 1, "name": "Rhino"},
        {"id": 2, "name": "Elephant"},
        [ ... ]
    ]
}
```

Map Sightings

Requires credentials: include *when making the request*

Endpoint: GET /discover/

Description:

Fetches all past sightings for the current user as well as all public sightings of other users to display on a map.

Response

On next page

```
{
    "success" : true,
    "message" : "Successfully fetched all discoveries",
    "data" : [
        {
            "image_url": "animal.png",
            "animals": ["Impala", "Warthog", ...],
            "geolocation_long": 24.35728,
            "geolocation_lat": -13.34566
        },
        { ... }
    ]
}
```

Sightings Service

Identify Animal

Requires credentials: include *when making the request*

Endpoint: POST /sightings/

Description:

Uses an AI model to identify animals in an uploaded image

Request Body

Content-Type: multipart/form-data

Field	Value	Description
file	*uploaded image*	User uploaded image
longitude	53.12345	Longitude of where the image was taken/uploaded
latitude	12.34567	Latitude of where the image was taken/uploaded

Response

```
{
    "success": true,
    "message": "Successfully created new sighting",
    "data": {
        "image_url": "animal.png",
        "animal": "Giraffe",
        ...
    }
}
```

Fetch Sighting

Endpoint: GET /sightings/:id

Description:

Fetches the details for the specified sighting

Response

```
{
    "success": true,
    "message": "Sighting retrieved successfully",
    "data": {
        "image_url": "animal.png",
        "animal": "Giraffe",
        ...
    }
}
```

Fetch User History

Requires credentials: include *when making the request*

Endpoint: GET /sightings/history

Description

Fetches all of the current user's past sightings

Response

```
{
    "success": true,
    "message": "Sighting history retrieved
successfully",
    "data": [
        {
            "image_url": "animal.png",
            "animal": "Giraffe",
            ...
        },
        { ... }
    ]
}
```

Architectural Requirements

Architectural Quality Requirements

#	Requirement	Reasoning	Expected Quantities
Q1	Reliability	Animals need to be correctly identified by the AI as often as possible	99.5% accuracy
Q2	Security	Protect user data, protect endangered animals,	roll-based access control
Q3	Performance	Allow user to get result as quick as possible, allow for sufficient amount of QPS	≥ 200 QPS
Q4	Scalability	Handle burst seasons such as holiday seasons (December, and summer in general)	25 requests per second
Q5	Availability	Detection should be available anywhere, Users should experience minimal downtime	on-device detection, 99% uptime
Q6	Extensibility	Allow for new features and animals to be added easily	N/A

Architectural Strategies

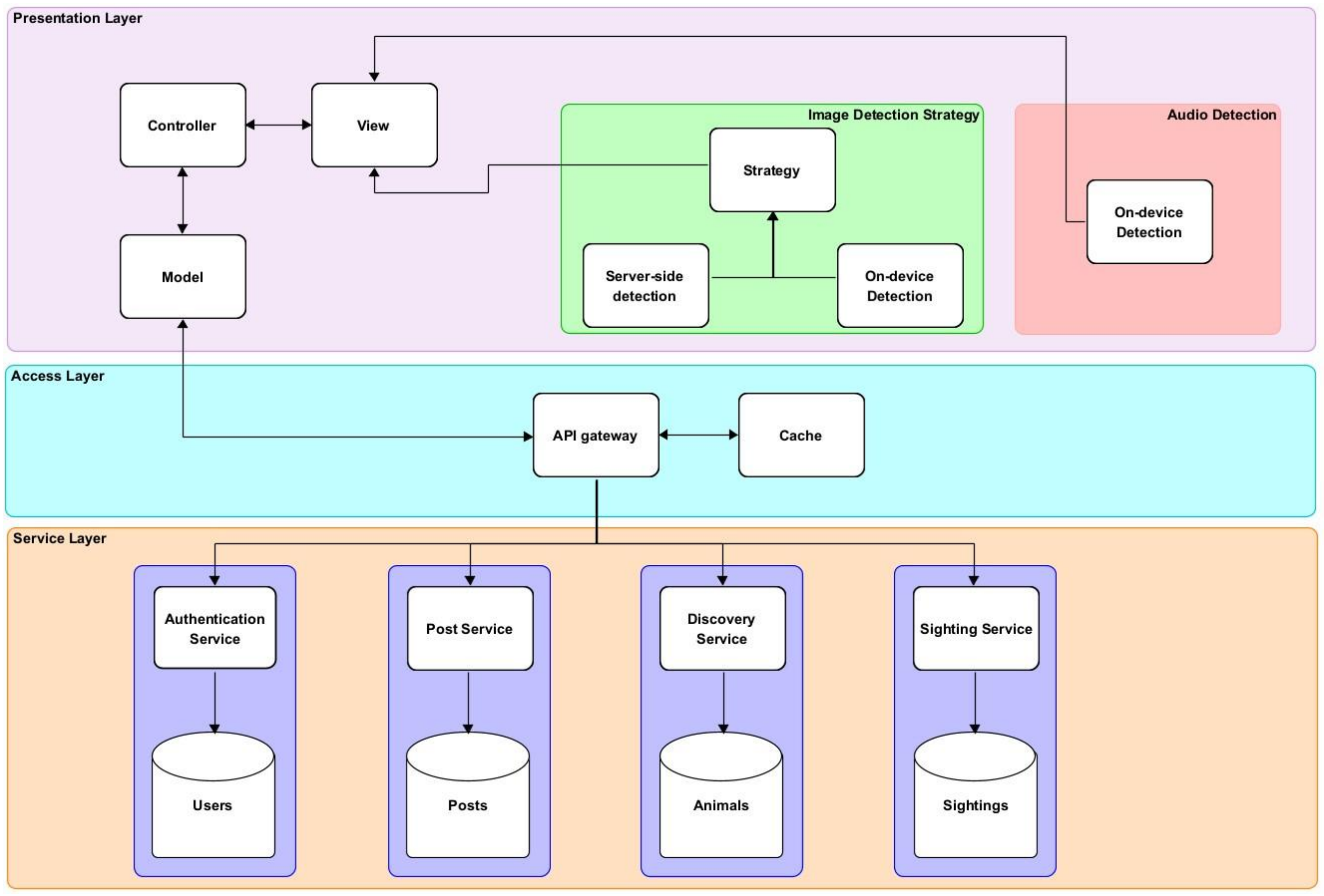
Requirement	Architectural Strategies
Reliability	Layered
Security	TLS & tokens
Performance	On-device inference (togglable)
Scalability	Horizontal Scale-out
Availability	Replication

Reasoning:

- **Reliability - Layered**
    - Since reliability requires an AI model with 99.5% accuracy, a layered approach is beneficial since the system is organized into layers. These layers promote separation of concerns and limits dependencies between layers, in turn, making the debugging and testing easier in isolation, further increasing the accuracy and reliability. Errors are contained which further simplifying their fixes

- **Security - TLS & Tokens**
    - TLS (Transport Layer Security)
        - Provides encrypted communication between the user and the system, preventing unauthorized interception.
    - Tokens
        - Used for authentication and authorization, supporting role-based access control enforcing only permitted users access to certain data and functionality.
- **Performance: On-device inference (togglable)**
    - Allows for faster queries by decreasing latency caused by data being transferred to and from to the server and client. It also reduces load on the central servers and reduces QPS achieving the target of "≥ 200 QPS", preventing bottlenecks.
- **Scalability: Horizontal Scale-out**
    - By adding more computing resources (horizontal scale-out), when the demand increases (burst seasons), it allows the distribution of the work load across multiple instances increasing performance during said peak-seasons.
- **Availability: Replication**
    - Through replication, the creation of copies of the data and components allows for redundancy preventing down-time since the copy took over from the failed component (whether its another server that takes over or a user's device)

Architectural Diagram



Architectural Design and Patterns

App Domains	Architectural Pattern
Frontend	MVC
AI	Event Driven and (strategy for toggle)
Backend	Micro services
API	API Gateway

Reasoning:

- **Frontend – MVC**
  - MVC ensures the interface, state management, and logic are separated. This makes the system easier to test and maintain, which improves **reliability** because issues in the UI won't affect the detection logic.
  - The clean separation also supports **extensibility**: new features like maps, sightings feeds, or moderation dashboards can be added without rewriting the entire app.
  - This works well with React, where components naturally represent Views and logic can be handled in Controllers or custom hooks.

- **AI – Event-Driven + Strategy (Performance, Scalability, Availability)**
    - An **event-driven approach** means every detection (image/audio) is queued for processing. This makes it easier to handle bursts of requests because the system can process them as resources become available, which supports **scalability** during peak seasons.
    - The **strategy pattern** allows toggling between on-device inference (fast, private, works offline) and cloud inference (more powerful, centralized). This improves **performance** by reducing latency when running locally, and it supports **availability** by allowing users to keep detecting animals even if the server is down or connectivity is poor.
- **Backend – Microservices**
    - Splitting the backend into small, independent services improves **scalability** because the most demanding services (like inference) can be scaled without affecting others.
    - This also improves **availability** since if one service fails (like notifications), the rest of the system still functions.
    - Microservices also make the app more **extensible** by allowing new detection services, data pipelines, or APIs to be added without disrupting existing ones.
- **API – API Gateway**
    - An API gateway provides a single entry point to the system, simplifying **security** with TLS encryption and token-based access control. It ensures that only authorized users can access sensitive data like the locations of endangered species.
    - The gateway makes the system more **extensible** because new APIs or versions can be introduced behind it without breaking existing clients.

Architectural Constraints

Technical constraints

• iNaturalist (Images) and Xeno-canto (Audio) datasets must be used for training the animal identification model • The system must adhere to the above-mentioned functional and quality requirements.

Business constraints

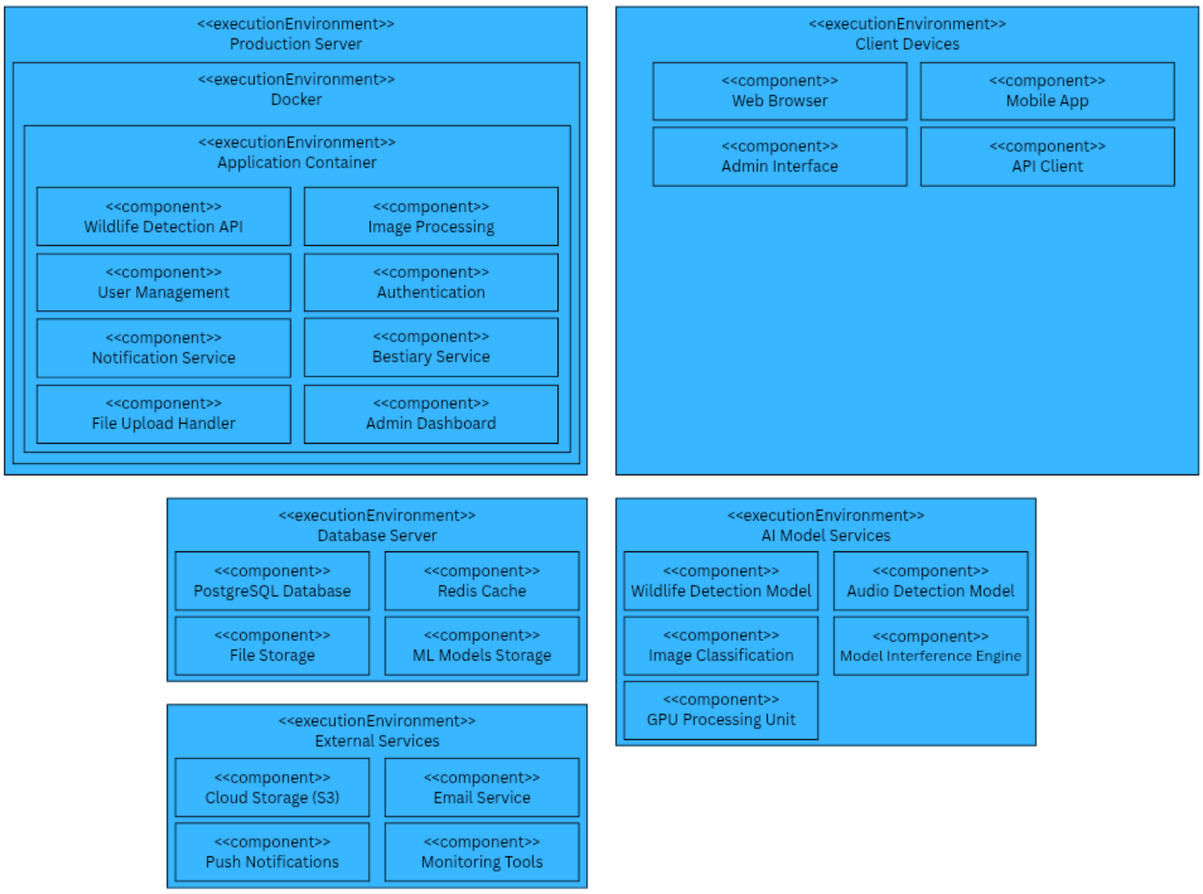
• Free and open-source services must be used as much as possible for cloud hosting, 3rd party services, etc. • A strict timeline must be followed to ensure in-time product delivery for demonstration sessions.

Security constraints

• Special attention must be given to protecting and obscuring the location data of species that are endangered and/or vulnerable to poaching to prevent malicious exploitation of the system.

Deployment Model

Deployment Model Diagram



<<executionEnvironment>> Production Server
<<executionEnvironment>> Docker
<<executionEnvironment>> Application Container
<<component>> Wildlife Detection API
<<component>> Image Processing
<<component>> User Management
<<component>> Authentication
<<component>> Notification Service
<<component>> Bestiary Service
<<component>> File Upload Handler
<<component>> Admin Dashboard
<<executionEnvironment>> Client Devices
<<component>> Web Browser
<<component>> Mobile App
<<component>> Admin Interface
<<component>> API Client
<<executionEnvironment>> Database Server
<<component>> PostgreSQL Database
<<component>> Redis Cache
<<component>> File Storage
<<component>> ML Models Storage
<<executionEnvironment>> AI Model Services
<<component>> Wildlife Detection Model
<<component>> Audio Detection Model
<<component>> Image Classification
<<component>> Model Interference Engine
<<component>> GPU Processing Unit
<<executionEnvironment>> External Services
<<component>> Cloud Storage (S3)
<<component>> Email Service
<<component>> Push Notifications
<<component>> Monitoring Tools

Overview

Our Wildlife Detection System employs a **blue/green deployment topology** utilizing **Render** as the primary cloud platform to ensure zero-downtime deployments and rapid rollback capabilities. The system is architected as containerized microservices with Render's native deployment capabilities, providing scalable and maintainable infrastructure for wildlife detection and monitoring.

Target Environment

Render Cloud Platform

- **Primary Platform**: Render (render.com)
- **Deployment Type**: Native Render services with Docker containers
- **Content Delivery**: Render's global CDN
- **Database**: Render PostgreSQL and Redis instances
- **Edge Computing**: Render's global edge network

Environment Tiers

1. **Production Environment**: Live system serving end users
2. **Preview Environment**: Automatic preview deployments for pull requests
3. **Development Environment**: Local development with Render CLI
4. **Staging Branch**: Dedicated staging deployments

Deployment Topology

Blue/Green Deployment Strategy on Render

The system implements a blue/green deployment pattern using Render's native features:

- **Blue Environment**: Current production services receiving live traffic
- **Green Environment**: New deployment with zero-downtime switching
- **Traffic Switching**: Render's instant traffic routing capabilities
- **Rollback Capability**: One-click rollback through Render dashboard
- **Health Checks**: Render's built-in health monitoring during deployments

Multi-Service Architecture on Render

Client Layer

- **Web Application**: Render Static Site for React/Vue frontend
- **Mobile API Gateway**: Render Web Service acting as API proxy
- **Admin Dashboard**: Separate Render Static Site deployment
- **Load Balancing**: Render's automatic load balancing across regions

API Gateway Layer

- **Primary API Gateway**: Render Web Service with Node.js/Express

- **Rate Limiting**: Implemented at application level
- **Authentication**: JWT-based auth with Render environment variables
- **SSL/TLS**: Automatic HTTPS with Render's managed certificates

Microservices Layer

Core Services (Each deployed as separate Render Web Services):

- **Wildlife Detection Service**: Express.js container on Render
- **User Management Service**: Express.js container on Render
- **Image Processing Service**: Express.js container with file upload handling

Supporting Services:

- **Notification Service**: Express.js service with email/SMS integration
- **Analytics Service**: Express.js service for data processing
- **Content Management Service**: Express.js service for media handling

Data Layer

- **Primary Database**: Render PostgreSQL (managed database)
- **File Storage**:
  - Render's ephemeral disk for temporary files
  - External object storage (AWS S3/Cloudinary) for persistent media
- **Backup Strategy**: Render's automated PostgreSQL backups

Monitoring & Operations Layer

- **Logging**: Render's native logging with log streaming
- **Metrics**: Render's built-in performance monitoring
- **Alerting**: Custom alerting through webhooks and external services
- **Health Monitoring**: Render's automatic health checks and restart policies

Containerization Strategy

Docker on Render

- **Dockerfile**: Optimized multi-stage builds for each service
- **Base Images**: Node.js official images or Alpine variants
- **Build Process**: Render's native Docker build pipeline
- **Registry**: Render's integrated container registry
- **Environment Variables**: Render's secure environment variable management

Infrastructure as Code

Environment Variable Management

- **Secrets**: Secure storage of API keys and credentials
- **Configuration**: Environment-specific settings
- **Integration Keys**: External service authentication tokens
- **Feature Flags**: Runtime configuration management

Deployment Pipeline

Git-based Automatic Deployment with Render

1. **Automatic Deployment Workflow**:
   - **Dev Branch**: Automatic deployment to development environment on successful push
   - **Main Branch**: Automatic deployment to production on successful push (no errors)
   - **Error Handling**: Deployments blocked automatically if build/test errors detected
   - **Pull Request Integration**: Preview deployments for code review
2. **Build Process**:
   - Automatic Docker image building on Render
   - Dependency caching for faster builds
   - Build-time environment variable injection
   - Automated security scanning
   - **Deployment Gates**: Only deploys if no errors in build/test phase
3. **Deployment Process**:
   - Zero-downtime rolling deployments
   - Health check validation before traffic switching
   - Automatic rollback on deployment failure
   - Deploy logs and monitoring
4. **Quality Gates**:
   - Build must complete without errors
   - Basic health checks must pass
   - Environment variables must be properly configured
   - Service dependencies must be available

Tools and Platforms

Development Tools

- **Render CLI**: Local development and deployment management
- **Docker**: Containerization and local testing
- **Git**: Version control with automated deployments

Monitoring and Observability

- **Render Metrics**: Built-in performance and resource monitoring
- **Custom Logging**: Application-level logging with log aggregation
- **External APM**: Integration with services like Datadog or New Relic
- **Uptime Monitoring**: External services for availability monitoring

Security

- **Environment Variables**: Secure credential and configuration management
- **HTTPS**: Automatic SSL/TLS certificate management
- **Network Security**: Render's built-in DDoS protection
- **Access Control**: Team-based access management in Render dashboard

Quality Attribute Support

Scalability

- **Horizontal Scaling**: Render's auto-scaling based on CPU/memory usage
- **Database Scaling**: Render PostgreSQL with connection pooling
- **CDN Scaling**: Global content delivery through Render's edge network
- **Regional Deployment**: Multi-region deployment capabilities

Reliability

- **High Availability**: Render's 99.95% uptime SLA
- **Auto-restart**: Automatic service restart on failures
- **Health Monitoring**: Continuous health checks with alerting
- **Geographic Redundancy**: Multi-region deployment support

Maintainability

- **Blue/Green Deployments**: Zero-downtime deployments with instant rollback
- **Preview Environments**: Automatic staging environments for every PR
- **Service Isolation**: Independent deployment and scaling of microservices

- **Configuration Management**: Centralized environment variable management

Security

- **Managed Infrastructure**: Render handles infrastructure security patches
- **Automatic HTTPS**: SSL/TLS certificates automatically managed
- **Environment Isolation**: Secure separation between environments
- **Access Logs**: Comprehensive logging for security auditing

Our deployment model leverages Render's managed platform capabilities while maintaining the blue/green deployment strategy and ensuring all quality requirements are met for the Wildlife Detection System.

Technology Choices

The following technologies are the preferred technologies for this project, selected by the team:

Use Case	Proposed Technology (or Framework) options
Frontend	React 18+ with Next.js, NextUI
Backend	Node.js & Express or Spring Boot
Database	PostgreSQL or MongoDB
API	REST or GraphQL
Documentation	Doxygen, Overleaf, Markdown and Google Docs
Version Control	Git, Github for repository hosting
Team Organization	GitHub Boards, Face-to-Face meetings and Discord
Testing	Jest and Cypress
Hosting	Vercel, AWS or GCP Cloud Services
AI	Python using YOLOv5, TensorFlow, PyTorch
CI/CD	GitHub Actions - GitGuardian, Build, Lint, Test and Deploy
IDE	VSCode

Frontend:

React

Overview

We initially began development using **React Native** to create a mobile application. However, as the project progressed, we identified a stronger need for a web-first experience with greater flexibility and easier integration of web technologies. This led us to transition to **React**, a powerful JavaScript library for building responsive and interactive user interfaces.

Why We Switched from React Native to React

- **Web-first focus**: Our primary target platform is the browser, making React a better fit for delivering a seamless web experience.
- **Simpler integration**: React offers better support for web-based libraries and APIs required for camera access, audio recording, and geolocation maps.
- **Easier testing and deployment**: Web apps built with React are easier to test, debug, and deploy using standard web tools and platforms.
- **Better accessibility and reach**: A browser-based app can be accessed across all devices without requiring installation, which aligns with our goal of broad user accessibility.

Pros of Using React

- Large ecosystem and community support
- Component-based architecture for reusable, maintainable UI
- Fast rendering with a virtual DOM
- Works seamlessly with Firebase for real-time updates
- Optimized for responsive design and cross-device compatibility

Fit for Our System

React is well-suited to our wildlife detection platform, allowing us to build a responsive, accessible, and performant web application. Its modular structure simplifies the development of complex features like camera integration, audio capture, and interactive maps, while maintaining a consistent experience across all devices.

Backend:

Node.js & Express

Overview

For our wildlife detection platform, we selected **Node.js with Express** as our backend framework. Node.js is a fast, event-driven JavaScript runtime, and Express is a minimalist web application framework that simplifies API development. This combination offers the flexibility, performance, and scalability needed for our system.

Why We Chose Node.js & Express

- **JavaScript across the stack** – Enables a consistent development experience and easier code sharing between frontend and backend
- **Non-blocking I/O** – Handles many concurrent requests efficiently, ideal for real-time updates (e.g., live sightings)
- **Extensive npm ecosystem** – Speeds up development with access to thousands of libraries and middleware
- **RESTful API support** – Express simplifies routing and API structuring for clean and maintainable backend logic
- **Fast development cycle** – Lightweight setup, quick prototyping, and easy deployment

Fit for Our System

Node.js with Express is an excellent match for our platform's real-time, user-interactive features. It enables rapid development of RESTful APIs, integrates smoothly with Firebase, and handles tasks like user management, data submission, and real-time notifications efficiently. Its performance and flexibility make it the ideal choice for supporting our responsive, scalable web application.

Database:

PostgreSQL

Overview

PostgreSQL is a powerful, open-source object-relational database system with a strong reputation for reliability, feature robustness, and performance.

Pros

- ACID compliance ensuring data integrity and reliability
- Excellent for structured data with complex relationships
- Strong support for geographic data types and operations
- Mature, enterprise-ready technology with extensive documentation
- Highly extensible with custom data types and functions

Fit

PostgreSQL's support for geographic data types makes it particularly well-suited for our geolocation mapping features. Its robust transaction support will ensure data integrity when handling concurrent user operations and sightings records.

AI:

YOLOv5, TensorFlow, PyTorch

Overview

For our core AI functionality, we'll be leveraging state-of-the-art frameworks and models:

1. **YOLOv5**: A real-time object detection system optimized for speed and accuracy
2. **TensorFlow/PyTorch**: Leading deep learning frameworks for building and training custom AI models

Pros

- YOLOv5 offers excellent performance-to-accuracy trade-offs for mobile applications
- TensorFlow provides production-ready deployment options including TensorFlow.js and TensorFlow Lite
- PyTorch excels in research and rapid prototyping of new AI models
- All solutions have strong community support and documentation
- Compatibility with transfer learning from pre-trained models

Fit

These technologies will power our core wildlife recognition capabilities. YOLOv5 will be optimized for real-time camera-based detection, while custom audio recognition models will be developed using TensorFlow or PyTorch. The selection between TensorFlow and PyTorch will be determined during the development phase based on specific model requirements and performance testing.

API:

Our API will be implemented using RESTful principles or GraphQL, depending on the specific needs identified during the development process:

- **REST API**: Simple, stateless architecture that's well-established and easy to implement
- **GraphQL**: Query language that allows clients to request exactly the data they need, reducing over-fetching

The final decision will balance factors including client-side data requirements, backend complexity, and team expertise.

Documentation, Version Control, Team Organization, Testing, and Hosting:

We will follow similar approaches to those outlined in the ABC Travel Planner proposal, utilizing industry-standard tools and practices including:

- **Documentation**: Overleaf, Markdown, Doxygen, and Google Docs
- **Version Control**: Git with GitHub repository hosting
- **Team Organization**: GitHub Project Boards, Discord, and face-to-face meetings
- **Testing**: Jest for unit tests and Cypress for end-to-end testing
- **Hosting**: Render for web application, and API hosting