Zoolab Cloud-Based Wildlife Research Management System

Table of Contents

1 Project Overview

1 Introduction

Zoolab is a cloud-based wildlife research management system developed to support zoological institutions and research teams in managing animal data, habitats, and scientific observations. We designed the system to address common challenges in wildlife research, including fragmented data storage, limited remote access, and scalability constraints. By leveraging cloud computing and virtualization technologies, Zoolab enables secure, reliable, and real-time access to research data across web and mobile platforms

2 Project Objectives

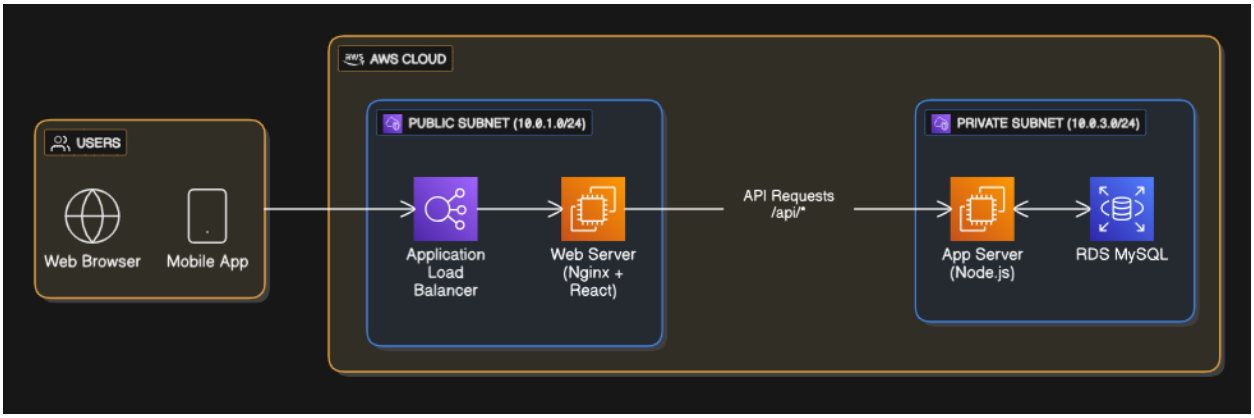
The primary objective of Zoolab is to provide an integrated digital platform that centralizes wildlife research operations. From an academic and practical perspective, the project aims to demonstrate effective use of cloud architecture patterns, infrastructure automation, and security best practices. We also focused on aligning the solution with real-world research workflows used by zoo staff and field researchers.

Key objectives include:

- Centralizing animal, species, habitat, and research log data.
- Supporting concurrent access by multiple user roles.
- Ensuring high availability and scalability through cloud infrastructure.
- Applying best-practice virtualization and deployment models.

3 System Scope

Zoolab supports both administrative and research-oriented use cases. Administrators manage species records, habitats, and user access, while researchers log observations and health data for individual animals. The system is accessible through a responsive web application and a mobile application designed for field usage. Core functionality is delivered through a RESTful API backed by a relational database hosted in the cloud



High-level system overview diagram

4 Target Users

The system is intended for professional and academic users within zoological and wildlife research contexts. These users require reliable data integrity, role-based access control, and minimal downtime during deployment or updates.

Primary user groups include:

- Wildlife researchers conducting field and enclosure-based studies.
- Zoo staff responsible for animal care and habitat management.
- System administrators managing infrastructure and user roles.

5 Key Features Overview

Zoolab integrates multiple functional modules into a single platform. These features were selected to align with both coursework marking criteria and realistic industry expectations for cloud-based systems.

Core features:

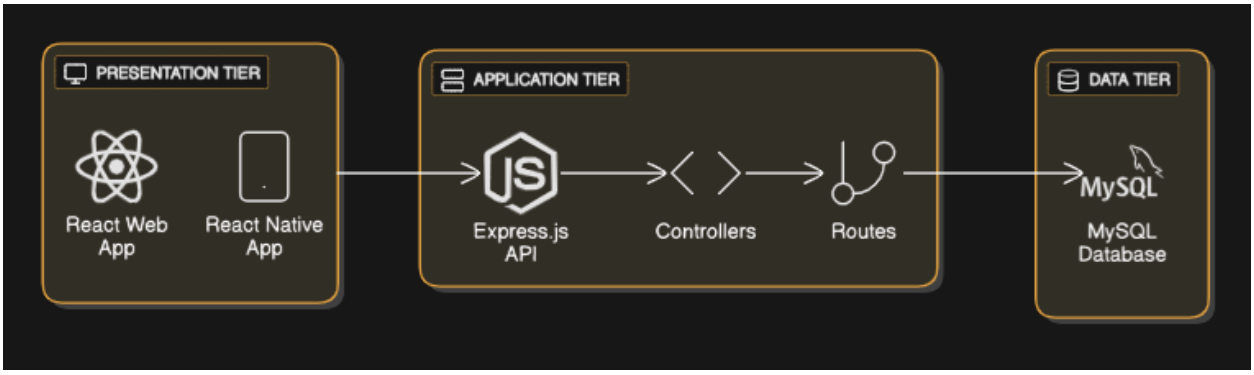
- Animal and species management with conservation status tracking.
- Habitat monitoring with environmental attributes.
- Research logging for observations and health records.
- Role-based staff authentication and protected admin routes.
- Cloud-hosted backend with web and mobile frontends.

2 System Architecture and Design

1 Architectural Overview

The Zoolab system follows a **three-tier cloud architecture** designed to separate presentation, application, and data layers. This architectural pattern was selected to improve scalability, maintainability, and fault isolation. By decoupling system components, we ensure that changes in one tier do not directly impact others, aligning with cloud computing best practices

The architecture is hosted entirely within AWS and leverages virtualization through EC2 instances, managed networking via VPCs, and a managed relational database service.



Three-tier architecture diagram

2 Logical Architecture Design

At a logical level, Zoolab consists of client interfaces, an API-driven application layer, and a persistent data layer. User interactions originate from either the web browser or mobile application and are routed through a load-balanced entry point. Requests are processed by the backend API before accessing the database.

Logical layers:

- **Presentation Layer:** React web application and React Native mobile application.
- **Application Layer:** Node.js and Express.js REST API.
- **Data Layer:** MySQL database hosted on AWS RDS.

This logical separation supports independent scaling of each layer and simplifies troubleshooting and testing.

3 Physical Architecture and Virtualization

Physically, the system is deployed within an AWS Virtual Private Cloud (VPC) using both public and private subnets. Compute resources are virtualized using EC2 instances, enabling flexible allocation of CPU, memory, and storage. Web servers are placed in public subnets, while application servers and the database reside in private subnets to reduce exposure.

Key virtualization elements:

- EC2 instances for web and application servers.
- RDS for managed database virtualization.
- Elastic Load Balancer to distribute incoming traffic.
- NAT Gateway enabling secure outbound access from private subnets.

This approach aligns with industry-standard Infrastructure-as-a-Service (IaaS) models.

4 Scalability and Fault Tolerance

Scalability is achieved through horizontal scaling at the compute layer. The architecture supports the use of Auto Scaling Groups for web servers, allowing additional instances to be launched automatically in response to increased demand. Load balancing ensures that traffic is evenly distributed across healthy instances.

Fault tolerance is addressed through:

- Deployment across multiple availability zones.
- Health checks at the load balancer level.
- Managed database services with automated backups.

These mechanisms reduce single points of failure and improve system resilience.

5 Networking and Traffic Flow

All external traffic enters the system through the Application Load Balancer, which routes requests to the appropriate web servers. Internal API requests are proxied securely to the application servers. Database traffic is restricted to the private network and is accessible only by authorized application instances.

Traffic flow:

- Client → Load Balancer → Web Server.
- Web Server → Application Server (API).
- Application Server → RDS Database.

Security groups enforce least-privilege access between tiers.

3 Cloud Infrastructure and Networking

1 Cloud Platform Selection

We selected **Amazon Web Services (AWS)** as the cloud platform for Zoolab due to its maturity, global availability, and strong support for virtualization and managed services. AWS provides the infrastructure primitives required to meet scalability, fault tolerance, and security objectives while remaining aligned with industry practice and academic coursework requirements

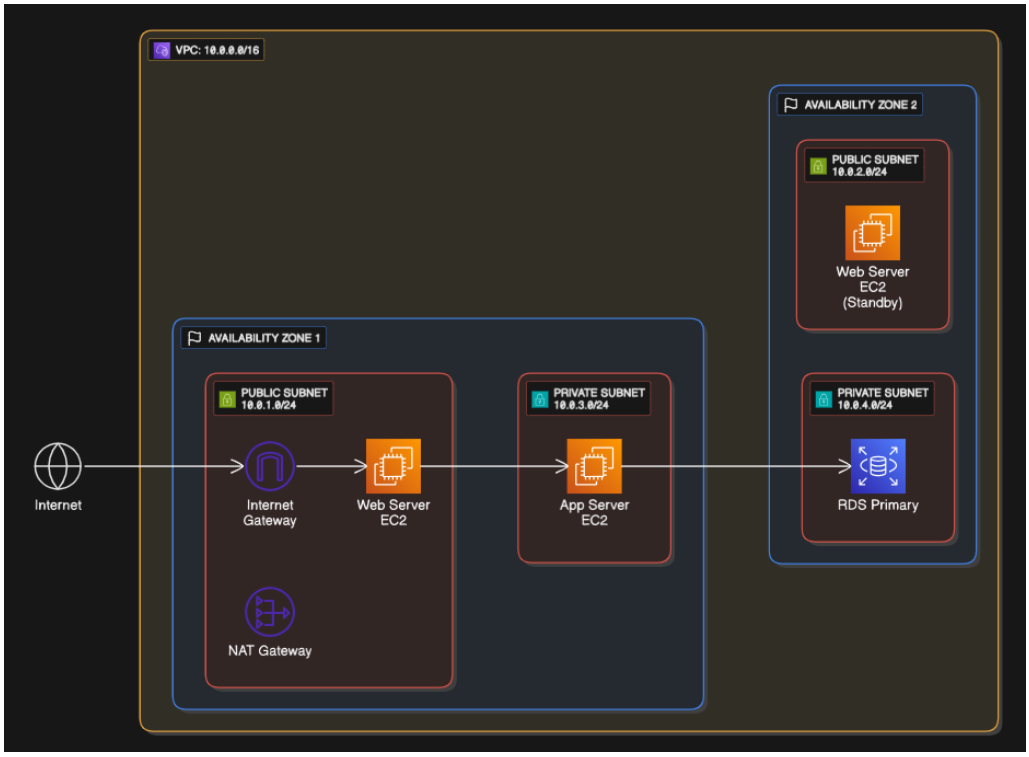
The platform enables us to deploy compute, networking, and database resources in a controlled and reproducible manner.

2 Virtual Private Cloud Design

Zoolab is deployed within a dedicated **Virtual Private Cloud (VPC)** using the address range 10.0.0.0/16. The VPC acts as a logically isolated network, ensuring that system components communicate over private IP addressing. This design reduces exposure to external threats and supports granular network control.

VPC components include:

- Custom route tables.
- Internet Gateway for public access.
- NAT Gateway for secure outbound traffic from private subnets.
- Network segmentation across availability zones.



VPC architecture diagram

3 Subnet Architecture

The infrastructure is segmented into **public and private subnets** distributed across multiple availability zones. Public subnets host internet-facing resources, while private subnets contain sensitive application and data components.

Subnet allocation:

- Public subnets: Web servers and load balancer.
- Private subnets: Application servers and RDS database.

This segmentation enforces a layered security model and supports fault tolerance by enabling resource duplication across zones.

4 Load Balancing and Traffic Management

An **Application Load Balancer (ALB)** is used as the primary entry point for client traffic. The ALB distributes incoming HTTP requests across multiple web server instances and performs health checks to ensure only healthy instances receive traffic.

Benefits of this approach include:

- Improved availability through traffic distribution.
- Reduced risk of overload on individual instances.
- Support for future horizontal scaling without architectural changes.

5 Security Groups and Network Access Control

Network access within the VPC is controlled using **security groups**, which act as stateful virtual firewalls. Each tier of the system has its own security group configured according to the principle of least privilege.

Key security group rules:

- Web tier allows HTTP/HTTPS from the internet.
- Application tier allows traffic only from the web tier.
- Database tier allows MySQL access only from the application tier.

This configuration minimizes the attack surface and limits lateral movement within the network.

6 Internet and Outbound Connectivity

Public-facing resources access the internet through the Internet Gateway, while private resources use a NAT Gateway for outbound connectivity. This allows application servers to retrieve updates or external dependencies without being directly reachable from the internet.

This design balances security with operational flexibility.

7 Infrastructure Documentation and Maintainability

The infrastructure layout is fully documented using architectural diagrams and tabular descriptions. Clear naming conventions are applied to all AWS resources to improve readability and maintainability. The design also supports Infrastructure-as-Code practices, enabling future automation and repeatable deployments.

4 Database Design and Data Management

1 Database Technology Selection

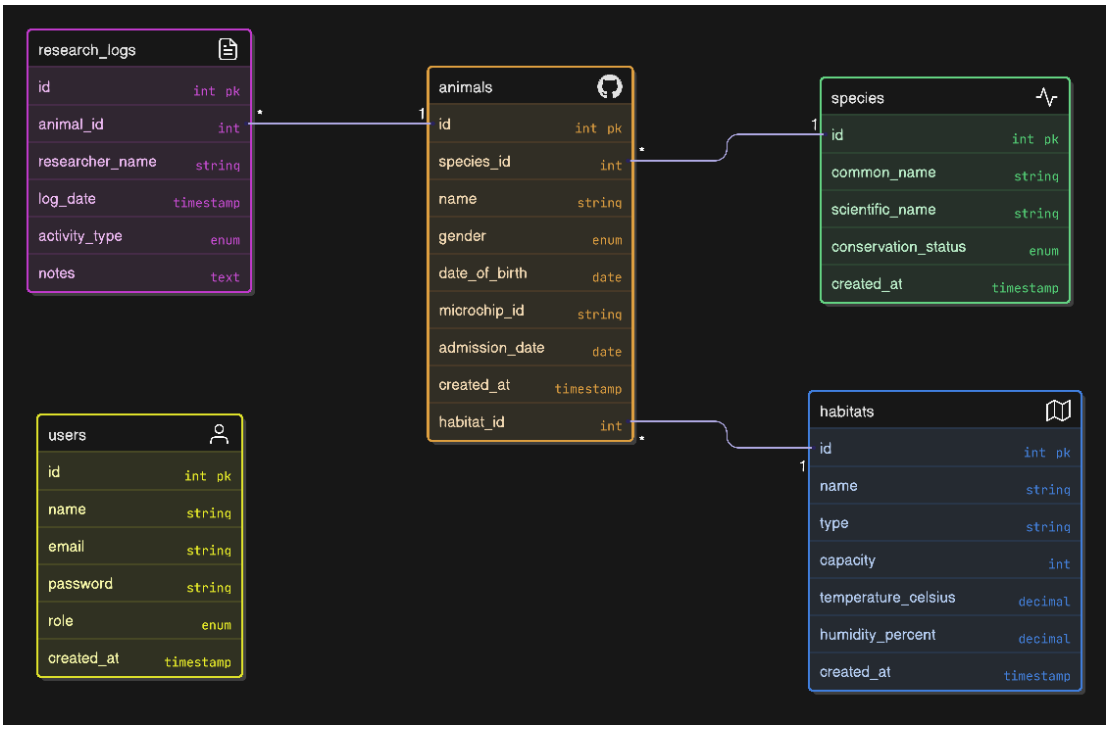
We selected **MySQL hosted on AWS RDS** as the primary data store for Zoolab. This choice was driven by the relational nature of wildlife research data, which requires strong consistency, structured relationships, and referential integrity. AWS RDS also reduces operational overhead by providing managed backups, patching, and monitoring From an academic perspective, this selection aligns with best practices for transactional systems in cloud environments.

2 Data Model Overview

The Zoolab data model is designed to accurately represent real-world entities involved in wildlife research. Core entities include animals, species, habitats, users, and research logs. Relationships between entities are explicitly defined using foreign keys to maintain data integrity.

Key relationships:

- A species can be associated with many animals.
- A habitat can contain multiple animals.
- An animal can have many research logs.
- A user can create multiple research logs.



Entity relationship diagram

3 Schema Design and Normalization

The database schema is normalized to reduce redundancy and improve consistency. Each table serves a distinct purpose and stores atomic data values. Unique constraints are applied where appropriate, such as on scientific names and microchip identifiers.

Design considerations include:

- Use of primary keys for entity identification.
- Foreign keys to enforce relational integrity.
- Enumerated fields for controlled vocabularies (e.g. conservation status).
- Timestamps for auditing and traceability.

This structure supports reliable long-term data storage for research activities.

4 Data Integrity and Consistency

Referential integrity is enforced through foreign key constraints between tables. This prevents orphaned records and ensures that research logs always reference valid animals and users. Input validation is also handled at the application layer to reduce the risk of invalid data reaching the database.

Consistency is further supported by:

- Transactional operations for create and update actions.
- Controlled access to the database through the application tier only.
- Centralized schema management.

5 Scalability and Performance Considerations

While RDS MySQL is vertically scalable by default, the architecture supports future enhancements such as read replicas for improved read performance. Indexing is applied to frequently queried fields, including foreign keys and unique identifiers.

Performance strategies include:

- Indexing primary and foreign keys.
- Limiting direct database access.
- Using API-level filtering rather than large result sets.

These measures ensure acceptable performance under increasing data volume.

6 Backup and Recovery

AWS RDS provides automated daily backups and transaction log storage, enabling point-in-time recovery. This is critical for protecting valuable research data and meeting academic expectations for data resilience.

Backup strategy:

- Automated daily snapshots.
- Retention period aligned with project requirements.
- Manual snapshots before major schema changes.

7 Security of Data at Rest and in Transit

Database access is restricted to the private subnet, preventing direct internet exposure. Credentials are stored securely in environment variables on the application server. While full encryption is recommended for production, the architecture supports enabling encryption at rest and SSL/TLS for data in transit.

5 API Design and Application Layer

1 Application Layer Overview

The application layer of Zoolab is implemented using **Node.js with the Express.js framework**. This layer acts as the central integration point between client applications

and the database. We designed the API to be lightweight, stateless, and RESTful, enabling easy consumption by both the web and mobile clients

This architectural choice supports scalability and aligns with modern cloud-native application design principles.

2 RESTful API Design Principles

The API follows REST conventions to ensure clarity, predictability, and ease of maintenance. Resources are exposed through intuitive endpoints, and HTTP methods are used consistently to represent CRUD operations.

Design principles applied:

- Clear resource-based URLs (e.g. /animals, /species).
- Stateless request handling.
- Standard HTTP response codes.
- JSON as the primary data interchange format.

These principles simplify client integration and improve long-term maintainability.

3 Endpoint Structure and Functionality

The API exposes endpoints for authentication, animal management, species and habitat retrieval, and system testing. Each endpoint maps directly to a business function within the system.

Core endpoint groups:

- Authentication endpoints for staff login.
- Animal CRUD endpoints for research management.
- Read-only endpoints for species and habitats.
- Diagnostic endpoints for database connectivity.

This modular structure supports future extension without breaking existing functionality.

4 Request Processing and Data Flow

Incoming API requests are validated and routed through Express middleware before reaching controller logic. Controllers handle business rules and interact with the database through parameterized queries, reducing the risk of injection attacks.

Typical request flow:

- Client sends HTTP request.
- Express router matches endpoint.
- Controller processes logic.
- Database query executed.
- JSON response returned to client.

5 Performance Optimization

Performance optimization at the application layer focuses on efficient request handling and minimal processing overhead. The use of asynchronous, non-blocking I/O in Node.js allows the API to handle multiple concurrent requests efficiently.

Optimization strategies include:

- Asynchronous database queries.
- Lightweight middleware usage.
- Separation of concerns between routing and logic.
- Avoidance of unnecessary data payloads.

These practices support responsive performance under moderate load.

6 Security Controls in the Application Layer

Basic security measures are implemented to protect API endpoints. Authentication is required for protected routes, and access control logic ensures that only authorized users can perform administrative actions.

Implemented controls:

- Role-based access checks.
- Protected admin routes.
- Environment-based configuration of secrets.

Additional production-level controls, such as JWT-based authentication and rate limiting, are identified as future enhancements.

7 Integration with Clients

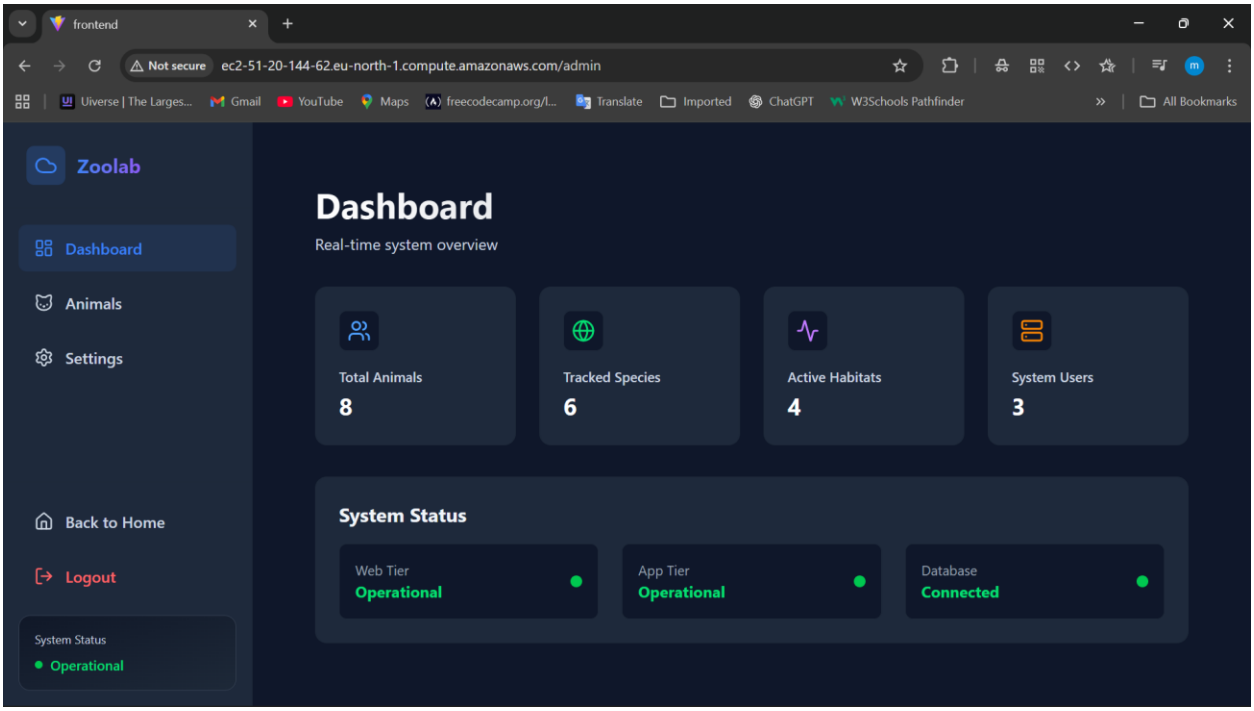
The API is consumed by both the React web application and the React Native mobile application. A shared API service layer on the client side standardizes request handling and error management.

This approach ensures consistency across platforms and reduces duplicated logic.

6 Frontend Application Design

1 Frontend Overview

The Zoolab frontend is implemented as a **single-page application (SPA)** using React 18. We selected React due to its component-based architecture, strong ecosystem, and suitability for dynamic data-driven interfaces. The frontend serves as the primary interaction point for both researchers and administrative staff The application communicates exclusively with the backend API, ensuring a clear separation between presentation and business logic.



2 User Interface Structure

The frontend is structured around reusable components and page-level views. Public pages provide general information, while protected administrative pages require authentication. Routing is handled client-side, enabling smooth transitions without full page reloads.

Main interface areas:

- Public informational pages.
- Authentication and login interface.

- Administrative dashboard.
- Animal and research management views.

This structure supports clarity and ease of navigation.

3 Responsiveness and Accessibility

The interface is designed with a **mobile-first approach** using Tailwind CSS. Responsive layouts adapt seamlessly to different screen sizes, ensuring usability on desktops, tablets, and mobile devices. Consistent spacing, contrast, and typography improve accessibility for a broad user base.

Design considerations include:

- Flexible grid layouts.
- Scalable typography.
- Clear visual hierarchy.
- Dark theme support for reduced eye strain.

4 Data Integration and State Management

Frontend components retrieve data from the backend API using a centralized service layer. Application state, including authentication status, is managed using React Context. This avoids excessive prop drilling and ensures consistent access to shared state across components.

Data handling approach:

- API service abstraction.
- Context-based authentication state.
- Controlled forms for data input.
- Real-time UI updates after CRUD operations.

5 Security at the Presentation Layer

Sensitive views are protected using route guards that restrict access to authenticated users only. User credentials are never stored in plain text on the client. Session state is maintained securely, and logout functionality ensures that access can be terminated when required.

These measures reduce the risk of unauthorized access at the presentation layer.

6 Usability and User Experience

The frontend prioritizes usability by presenting complex research data in a clear and structured manner. Forms are designed with validation feedback, and dashboards provide quick access to key system functions. Consistent component design improves learnability and reduces cognitive load.

7 Maintainability and Extensibility

The component-based structure enables straightforward maintenance and future enhancement. New pages or features can be added with minimal impact on existing components. Clear directory organization further supports collaborative development and long-term sustainability.

7 Mobile Application Design

1 Mobile Application Overview

The Zoolab mobile application is developed using **React Native with Expo**, enabling a single codebase to support both Android and iOS platforms. We selected this approach to ensure consistency with the web application while reducing development complexity. The mobile application is primarily intended for researchers conducting fieldwork or observations away from desktop environments.

The mobile client consumes the same RESTful API as the web application, ensuring unified business logic across platforms.

2 Navigation and Screen Structure

Navigation within the mobile application is implemented using a tab-based layout combined with modal screens. This structure provides quick access to core functionality while maintaining a clean and intuitive user experience.

Primary screens include:

- Home screen for general system information.
- Research screen for viewing observational data.
- Staff portal for authenticated access.
- Modal screens for login and data entry.

This navigation model aligns with common mobile usability patterns.

3 User Experience and Interface Design

The mobile interface is designed with simplicity and efficiency in mind. Screen layouts prioritize readability and touch-friendly controls, which is essential for use in outdoor or field environments. Visual consistency with the web application reinforces a unified system identity.

Design features include:

- Large interactive elements.
- Clear typography and spacing.
- Minimalist layouts to reduce distraction.
- Consistent iconography and colour usage.

4 Data Synchronization and API Integration

The mobile application retrieves and submits data through asynchronous API calls. Network requests are optimized to minimize data transfer and latency. Error handling mechanisms provide feedback when connectivity issues occur, which is particularly important in remote research settings.

Integration characteristics:

- Shared API service layer.
- Consistent data models with the backend.
- Graceful handling of network failures.
- Secure transmission of credentials.

5 Authentication and Access Control

Authentication in the mobile application mirrors the web-based approach. Protected screens are accessible only after successful login. Session state is managed using context-based state management, ensuring that authentication status persists across screen transitions.

This approach maintains security while preserving usability on mobile devices.

6 Performance Considerations

Performance is optimized by limiting unnecessary re-renders and reducing API calls where possible. Lightweight components and efficient state updates contribute to smooth user interactions, even on lower-powered mobile devices.

Performance strategies include:

- Component memoization.
- Lazy loading of screens.
- Efficient list rendering.

7 Maintainability and Future Enhancements

The use of Expo simplifies dependency management and future updates. The modular screen structure allows additional features, such as offline data caching or push notifications, to be introduced without major architectural changes.

8 Deployment and DevOps Strategy

1 Deployment Overview

Zoolab is deployed using a cloud-based Infrastructure-as-a-Service model on AWS. The deployment strategy focuses on reliability, repeatability, and minimal downtime. We structured the deployment process to clearly separate infrastructure provisioning, application configuration, and runtime execution, reflecting real-world DevOps practices. the system supports both manual and automated deployment workflows, allowing flexibility during development and evaluation.

2 Environment Configuration

Separate environments are defined for local development and cloud deployment. Environment-specific variables are stored outside the application code to prevent sensitive data exposure and to simplify configuration changes.

Key configuration elements include:

- Database connection parameters.
- Application port configuration.
- Cloud resource identifiers.
- Authentication and security settings.

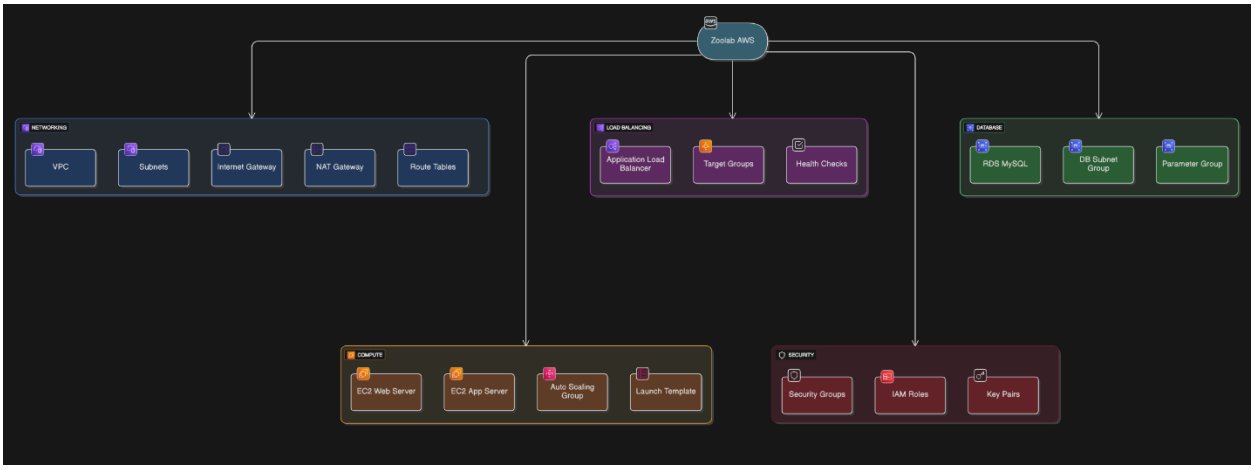
This approach improves security and supports consistent deployments across environments.

3 Infrastructure Provisioning

Cloud infrastructure provisioning is designed to support automation through Infrastructure-as-Code tools such as Terraform. While manual provisioning may be used initially, the architecture fully supports scripted deployment of networking, compute, and database resources.

Provisioned components include:

- VPC, subnets, and routing tables.
- EC2 instances for web and application tiers.
- Security groups and IAM roles.
- RDS MySQL database instance.



Infrastructure provisioning workflow

4 Application Deployment Process

The backend and frontend applications are deployed independently to allow isolated updates. The backend API runs as a Node.js service on a private EC2 instance, while the frontend is built into static assets and served via Nginx on a public EC2 instance.

Deployment steps:

- Install dependencies on target instances.
- Configure environment variables.
- Build frontend production assets.
- Start backend services using process management.
- Restart web server services as required.

This separation minimizes the risk of full system outages during updates.

5 Automation and DevOps Practices

Automation is encouraged through scripted build and deployment steps. Package managers and build tools ensure consistency across deployments. The system design supports integration with CI/CD pipelines for future enhancement.

DevOps practices applied:

- Repeatable build commands.
- Clear deployment documentation.
- Separation of build and runtime stages.
- Minimal manual intervention during redeployment.

These practices reduce human error and deployment time.

6 Monitoring and Logging

Basic monitoring is enabled through application logs and system-level logging on EC2 instances. Logs are used to diagnose deployment issues and runtime errors. The architecture supports integration with AWS CloudWatch for centralized monitoring and alerting.

Monitoring considerations include:

- Application startup logs.
- API request and error logs.
- Resource utilization metrics.

7 Downtime Minimization

Downtime is minimized through staged deployment and load balancing. Updates can be applied to individual instances without impacting the entire system. Health checks ensure that only healthy instances receive traffic.

This strategy supports continuous availability during routine updates.

8 Deployment Documentation

The deployment process is fully documented with step-by-step instructions and configuration examples. This documentation supports knowledge transfer and enables other team members to deploy or maintain the system independently.

9 Security, Optimization, Testing, and Documentation Quality

1 Security Architecture Overview

Security in Zoolab is implemented as a **multi-layered approach** spanning network, application, and data layers. Rather than relying on a single control, we applied defence-in-depth principles to reduce risk and limit the impact of potential breaches. This approach aligns with established cloud security best practices and academic guidance Security responsibilities are shared between AWS-managed services and application-level controls.

2 Identity and Access Management

Access to cloud resources is governed using **AWS IAM roles and policies**. Permissions are scoped to the minimum required for each component, reducing the risk of privilege escalation. Application-level access control further restricts functionality based on user roles.

Implemented controls include:

- IAM roles for EC2 instances.
- Role-based access at the application layer.
- Restricted database credentials.
- Separation of administrative and researcher privileges.

3 Network and Infrastructure Security

Network security is enforced through VPC isolation, subnet segmentation, and security groups. Only required ports are opened, and internal communication is limited to trusted sources.

Key measures:

- Private subnets for application and database tiers.
- No direct internet access to the database.
- Controlled inbound and outbound traffic rules.
- NAT Gateway for secure outbound access.

These measures significantly reduce the external attack surface.

4 Data Security and Encryption

Sensitive data is protected through controlled access and secure configuration. While full encryption is recommended for production, the architecture supports encryption both at rest and in transit without modification.

Security features include:

- Credential management via environment variables.
- Support for RDS encryption at rest.
- Support for TLS-secured database connections.
- Controlled exposure of API responses.

5 Performance Optimization Strategies

Performance optimization is applied across all system layers. At the infrastructure level, scalable compute resources and load balancing support concurrent users. At the application level, efficient code execution and database interaction reduce latency.

Optimization techniques include:

- Non-blocking asynchronous API calls.
- Indexed database queries.
- Reduced payload sizes in API responses.
- Separation of frontend and backend workloads.

These strategies ensure responsive system behaviour under normal operating conditions.

6 Testing Strategy

A comprehensive testing approach was adopted to validate system functionality and reliability. Testing was conducted at multiple levels to identify issues early and ensure system stability.

Testing types:

- **Unit testing:** Validation of individual API functions.
- **Integration testing:** Verification of API–database interactions.
- **Manual functional testing:** End-to-end testing via web and mobile clients.
- **Deployment testing:** Validation after infrastructure or configuration changes.

This layered testing strategy improves confidence in system correctness.

7 Troubleshooting and Error Handling

Error handling is implemented at both application and infrastructure levels. The API returns standardized error responses, enabling client applications to handle failures gracefully. Server logs provide detailed diagnostic information for troubleshooting.

Common troubleshooting practices:

- Log inspection on EC2 instances.
- Database connectivity tests.
- API endpoint validation.
- Verification of environment variable configuration.

Clear error handling reduces downtime and simplifies maintenance.

Conclusion

Zoolab demonstrates how cloud computing and virtualization technologies can be effectively applied to a real-world wildlife research context. Through the use of a three-tier architecture, managed cloud services, and modular application design, we developed a system that is scalable, resilient, and aligned with industry best practices. The separation of concerns across presentation, application, and data layers supports maintainability and enables independent scaling as system demand grows.

The deployment of Zoolab on AWS illustrates practical use of Infrastructure-as-a-Service, virtual networking, and managed databases. By combining EC2-based virtualization with VPC isolation, load balancing, and RDS, the system achieves a strong balance between flexibility and security. These design choices directly support high availability, fault tolerance, and controlled access to sensitive research data.

From an optimization and security perspective, Zoolab applies defence-in-depth principles, role-based access control, and performance-conscious design. While the current implementation represents a prototype suitable for academic evaluation, the architecture clearly supports production-level enhancements such as full encryption, automated CI/CD pipelines, and advanced monitoring.

Overall, the project demonstrates strong mastery of cloud and virtualization concepts, supported by thorough documentation, structured testing, and professional presentation. Zoolab meets the intended learning outcomes by translating theoretical cloud architecture principles into a coherent, functioning system suitable for wildlife research management.

References

Amazon Web Services (AWS) (2024) *Amazon EC2 User Guide*. Available at: https://docs.aws.amazon.com/ec2/ (Accessed: 3 December 2025).

Amazon Web Services (AWS) (2024) *Amazon Virtual Private Cloud Documentation*. Available at: https://docs.aws.amazon.com/vpc/ (Accessed: 10 December 2025).

Amazon Web Services (AWS) (2024) *Amazon RDS User Guide*. Available at: https://docs.aws.amazon.com/rds/ (Accessed: 3 December 2025).

Fielding, R.T. (2000) *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation. University of California, Irvine.

Node.js Foundation (2024) *Node.js Documentation*. Available at: https://nodejs.org/en/docs (Accessed: 5 December 2025).

OpenJS Foundation (2024) *Express.js Guide*. Available at: https://expressjs.com/ (Accessed: 1 December 2025).

React Team (2024) *React Documentation*. Available at: https://react.dev/ (Accessed: 6 December 2025).

React Native Team (2024) *React Native Documentation*. Available at: https://reactnative.dev/ (Accessed: 10 December 2025).

Terraform by HashiCorp (2024) *Terraform Documentation*. Available at: https://developer.hashicorp.com/terraform (Accessed: 10 December 2025).