Internal Records API

Serverless Infrastructure Project (AWS)

1. Project Overview

This project implements a serverless internal records service using AWS managed services. The system is designed to support internal operational workflows that require reliable record storage and retrieval without the overhead of managing servers.

The project focuses on building something small but real: a backend service that works end-to-end, scales automatically, logs its behaviour, and can be audited and monitored in a production setting.

The service is intended for internal use cases only, such as operational logging, system checks, administrative records, or lightweight workflow tracking.

2. Problem Statement

Organizations frequently need internal systems to track operational events and system activity. These systems are not customer-facing, but they still need to be dependable, observable, and easy to maintain.

Traditional server-based solutions often introduce unnecessary complexity for this type of workload. They require ongoing infrastructure management, capacity planning, and maintenance, even when usage is low or unpredictable.

The problem this project addresses is:

How can an internal records service be built in a way that is reliable, auditable, and cost-efficient, while minimizing operational overhead?

3. Solution Approach

To solve this problem, the project uses a fully serverless architecture built on AWS managed services. The design removes the need to provision or maintain servers and instead relies on event-driven components that scale automatically.

The system exposes a simple API for interacting with records, processes requests through backend logic, stores data in a managed database, and captures logs and audit information as part of normal operation.

This approach prioritizes simplicity, transparency, and operational correctness over unnecessary feature complexity.

4. Architecture Overview

At a high level, the system works as follows:

1. An HTTP request is sent to the service
2. Amazon API Gateway receives the request and routes it
3. AWS Lambda executes the request logic
4. Amazon DynamoDB stores or retrieves the record data
5. A response is returned to the caller
6. Logs and audit data are captured automatically

All components are managed services, which means scaling, availability, and fault tolerance are handled by AWS.

5. AWS Services Used

The project uses the following AWS services, each for a specific purpose:

- **Amazon API Gateway**
  Provides the HTTP interface and request routing for the service.
- **AWS Lambda**
  Executes backend logic in a stateless, event-driven manner.
- **Amazon DynamoDB**
  Stores internal records in a scalable, low-latency NoSQL database.
- **Amazon CloudWatch**
  Captures logs and runtime information used for debugging and monitoring.
- **AWS CloudTrail**
  Records API activity and configuration changes for audit purposes.
- **AWS Budgets**
  Monitors usage and helps prevent unexpected costs.

6. Data Model

Each record stored in DynamoDB includes the following attributes:

- recordID – unique identifier for the record
- createdAt – timestamp of record creation
- submittedBy – source of the record
- type – record classification
- status – current state of the record
- message – optional additional context

The schema is intentionally flexible so the system can support multiple internal use cases without structural changes

7. API Endpoints

GET /records

This endpoint retrieves all stored internal records.

The response is returned as structured JSON and is suitable for integration with internal dashboards or other services.

The API surface is intentionally minimal to keep the system easy to reason about and extend later if needed.

8. Configuration and Deployment

The Lambda function is configured using environment variables, which are used to reference infrastructure resources such as the DynamoDB table name. This avoids hardcoding values into the code and allows configuration changes without modifying application logic.

Code and configuration updates are deployed through the standard AWS Lambda deployment process to ensure changes are applied consistently.

9. Logging, Monitoring, and Auditing

Operational visibility was a core consideration during development.

- **CloudWatch Logs** capture Lambda execution details and error messages. These logs were actively used during development to diagnose runtime issues and configuration errors.
- **CloudTrail** records API-level activity and configuration changes, providing an audit trail of actions taken within the AWS environment.

Together, these services provide clear insight into how the system behaves and how it changes over time.

10. Cost Management

Cost awareness was built into the project from the start.

- AWS Budgets were configured to monitor monthly usage.

- The serverless architecture minimizes idle costs by eliminating always-on resources.

This approach aligns with best practices for internal systems that may have variable or low usage.

11. Security Considerations

The system follows basic security best practices:

- IAM permissions are scoped to only what the Lambda function requires.
- No public data storage is exposed.
- No sensitive or regulated user data is handled.
- Audit logging is enabled through CloudTrail.

The service is intended for trusted internal environments.

12. Challenges and Resolution

During development, several real-world issues were encountered, including environment variable mismatches and deployment sequencing problems. These issues resulted in runtime errors that were diagnosed using CloudWatch Logs.

Resolving these problems required careful inspection of logs, verification of configuration, and controlled redeployment. This process closely reflects how production issues are identified and resolved in real systems.

13. Known Limitations

- No authentication or authorization is implemented
- No pagination or filtering is applied to record retrieval
- The system is intended for internal use only

These limitations are acceptable given the current project scope.

14. Future Enhancements

Potential next steps include:

- Adding authentication and access control
- Supporting additional CRUD operations
- Implementing pagination and filtering
- Integrating with an internal user interface
- Adding structured metrics and alarms

15. Project Significance

This project demonstrates the ability to design and implement a production-style **serverless backend**, apply AWS best practices, and debug real runtime issues using cloud-native tooling.

Rather than focusing on feature breadth, the project emphasizes correctness, observability, and operational realism.

Final Summary

This project delivers a generic, reusable internal records service built on AWS serverless infrastructure. It provides a reliable and auditable foundation for internal operational workflows while keeping operational complexity and cost low.