**SITEVISION**

Lawal Harafa

**ABSTRACT**

This project focuses on the development and deployment of a cloud-based web project using AWS services. The project dubbed site-vision is a web-based incident logging software used to record incidents that happen on the construction site. This software can be used in both large and small-scale construction sites to ensure that all site activities are logged helping in site management and keeping track of incidents.

The project integrates AWS lambda for functions which are the logic behind the backend of our application. The functions are written in python majorly using boto3 library to perform the integrations. The data is stored in DYNAMODB database which is part of AWS service stack for development and deployment of cloud native applications. AWS API Gateway is used to direct our HTTP requests to channel the correct requests to the subsequent logic before fetching, updating or deleting data from the database.

Authentication and authorization to access the data is handled by AWS Cognito which is an authentication service. The website frontend is written in HTML, CSS and vanilla JavaScript and hosted in Amplify which is a service for hosting static or frontend services to consume other AWS services that are synonymous to the backend.

The main purpose of the project is to ensure traceability of site incidents and aggregation can lead to identification of the most prone ones. The user of the developed cloud-based web application should be able to create incidents by filling a form of date, title and description of the incident. The incident is saved to our database and a list of incidents is available for purposes of keeping track of what’s happening on site. This report details the design and development of the solution with strict adherence to the provided guidelines.

**INTRODUCTION**

The motivation behind the project is the need to keep track of incidents within construction sites. Given the sensitive nature pertaining safety of such a site such solutions come in handy to ensure all site incidents are logged and monitored. This data is crucial in ensuring that there is no repeat of the bad incidents and also corrective or compensating measures are put in place to reduce further escalation of the nature of the incident. Many companies rely on in-house solutions which are developed by individuals with coding experience but no background in such environments. Developing scalable and maintainable solutions that can be accessed anywhere is the way to go. In our case, we leverage technology that has proven to fit such requirements.

Construction projects often involve various dangerous operational challenges. This is due to the nature of the work that is undertaken that involves use of machinery that if misused could lead to bodily harm of the people using them. Keeping tabs of incidents will lead to the identification of repeat incidents and finding solutions to minimize them. It also helps with ensuring the company meets regulatory and compliance requirements to avoid reputational damage. Traditional approaches have proven to be slow, error-prone and inefficient with manual intensive solutions and little room for analytics for correlation purposes. Site-vision has therefore been developed as a lightweight solution for easy creation and retrieval of site incidents. This lightweight and serverless application is designed for real time retrieval to ensure on the spot analysis of repeat incidents and also a promise for predictive features that minimize such incidents by provision of the appropriate solutions.

The main objectives of the project are to ensure the design and development of a simple web interface for creation and retrieval of site incidents. The properties of the incident that are being logged are the title, date, location and description of an incident. These properties are enough to adequately analyze and give feedback on ways to curb a repeat of the logged incidents. We also aim to develop a scalable and maintainable serverless backend using AWS services. We choose services that have been tried and tested and focus on ways of ensuring our services work together coherently. We are also tasked with the objective of developing a library which will provide functionality to our already developed serverless application. In our case, we use python to create helper functions that will validate the incidents and ensure that they are in a format that is acceptable in our database and all required fields are filled out to avoid having garbage data. Having valid data makes it easier for analysis and also identification based on the date of the incident and making informed business, design and management decisions.

The project also aims to provision the web -based application in a way that we can make changes on the fly through continuous integration and deployment pipelines. This is achieved through use of services such as amplify in combination with GitHub to ensure that with each commit the changes are directly reflected on our site. This approach also ensures that we can keep track of our development progress and revert back to previous commits incase we break our production environment.

Site-vision aims to demonstrate use of cloud-based services in the resolution of a real-world problem. This is done through the application of cloud computing principles through use of modern tools that are provisioned by a trusted cloud provider such as AWS.

**PROJECT REQUIREMENTS**

The project requirements for our cloud-based solution are split into functional and non-functional requirements. We discuss the functional requirements of the project first.

Functional requirements of the project are actionable processes that a user can follow to use our application. The user should be able to submit a new incident via a web form which should be designed in a manner that is easy to use. The required fields to submit an incident are title, date, location and description of the incident being logged. We should be able to automatically assign an incident ID which ensures that the IDs don’t collide. This is done through randomization of the IDs ensuring that no single incident have the same ID. This will lead to confusion or even fatal errors that could affect the functioning of our application in extreme cases. Upon validation of the fields required to log an incident the application should be able to store the data provided in a NoSQL database which is part of our serverless backend provisioned by AWS. The application also provides an interface where we can retrieve and display all incidents as well as a button to delete an incident in case of erroneous entry of an incident.

A library is part of the functional requirements and it provides validation functionality. We ensure that the required fields; incident ID, title, location and description are captured before trying to make any computation involving sending data to the database.

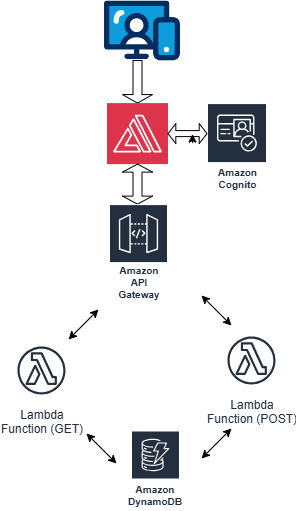
We provide authentication and authorization services to our application using AWS Cognito. The user should be able to register to our site and login to be able to access the restricted functional requirements within our application.

Non-functional requirements are associated with technical constraints and mostly affected by the project requirements. They mostly advice on the technologies to be used and affect the choice of cloud services. One of the non-functional requirements is that the project should be hosted on a cloud platform with the choice for this project being AWS. The website should also be responsive and mobile-friendly which ensures accessibility across a range of devices meaning our solution will be a versatile one. The backend of the application should also be stateless and event driven which is implemented by design across AWS provisioned services. The events that drive the backend are user inputs and submissions which are crucial in triggering the lambda functions which hold majority of our backend logic.

Data retrieval latency should be minimized which is handled internally by AWS and also in the code by ensuring we implement non-expensive retrieval computations. Low latency ensures our application is responsive and the user enjoys a great user experience. The implemented code should also be maintainable such that it follows language coding standards in terms of naming conventions and code organization. Implementation of modular architecture is also emphasized upon so that we can decouple each service to avoid failure of the whole system due to failure of a single service. The above discussed are the non-functional requirements and are aimed at ensuring longevity, scalability and maintainability of the code used in the development of the project.

**ARCHITECTURAL DESIGN**

The architecture design of the cloud-based web application is as shown below



The project consists of the above AWS services as described in the architecture diagram. The first illustration represents the user who uses the frontend of the application to make requests. This involves navigation across the web application either through authentication or submitting a new incident. The files that power the web interface are deployed on AWS amplify. This includes HTML, CSS and JavaScript files which provide the functionality to the user. Images and other supporting files are hosted here.

Before giving access to incidents, we must authenticate users and ensure that they have the proper authorization to perform actions within our application. AWS Cognito is an excellent lightweight service to provision such features without the need to manage users and create complex backend. It involves creation of user or identity pools and using common authentication mechanisms such as emails and passwords to provide authorization. We provision links to redirect to upon authentication which align to the logic of our application appropriately. In our case, pages containing incidents and creation of incidents are protected and dependent upon authentication and authorization from Cognito.

Amazon API Gateway acts as the handler of our API requests. Each request is associated with a particular function from retrieving all incidents using a GET request and submitting an incident via a POST request. The API Gateway service handles each request and directs it to the appropriate lambda function to handle it depending on the nature of the request. The main function of this service is to forward each of the request to the appropriate lambda function.

The lambda function service handles each unit backend logic that we have. We have logic to handle creation of an incident with the appropriate python code hosted in the lambda function to insert into our database. The other logic that is available is for retrieving all incidents and returning a JSON response consisting of all the incidents that have been submitted by a particular user. Lambda functions are the building blocks of our backend logic and we can create as many lambda functions as our functional requirements allow. They also provide a modularized architecture which enables as to separate each functional block of code based on the logic or functionality it provides. Lambda functions allow backend code to be executed on demand without server provisioning as a requirement.

Dynamo DB service provides NoSQL storage service of our data. We programmatically add values via our logic code that is saved in lambda functions. The table schema can be accessed separately and we can modify each of the attributes to match whatever input we want to obtain from the users. This service acts as the primary storage of data and is the most low-level service as per our architecture diagram. Dynamo DB contributes towards our project by provision of consistent fault tolerant storage for our data.

**CLOUD-SERVICE ANALYSIS**

The web application utilizes five AWS. They are Amplify, Cognito, API Gateway, Dynamo DB and lambda functions. Despite availability of other services that fit the task to be handled the above services were chosen for various reasons.

Amplify has been used for deployment of static files that run the frontend services of the web application. It also provides continuous integration and deployment via GitHub ensuring changes take effect once we make commits. Its advantages range from ease of deployment of files that power frontend applications by connecting to GitHub easily. Custom domain mapping and scalability is also provided out of the box. Its disadvantages are that it is less customizable in backend heavy application and provides limited configuration compared to other services such as S3 and CloudFront. Amplify was chosen for ease of integration with frontend services and GitHub for CI/CD making it easier to work with out of the box.

API Gateway was used to provide endpoints for the frontend to communicate with lambda functions. It is easy to integrate with Cognito and Amplify for scalable backends and robust functionality. It has built in security features such as logging and monitoring features for throttling to reduce wastage of our resources. Its disadvantage is that it requires more configuration to access the other features such as throttling and security monitoring. We settled for this service as it provides a consistent scalable way to expose our backend functionality coupled with security and throttling features.

Lambda functions have been used to run backend logic to store and retrieve incidents. They provide automatically scalable serverless backends meaning that there will be no infrastructure to manage. This service provides support for most of the programming languages out of the box making it easy to natively integrate with other AWS services. Its downside is that there may be cold start latency and also some execution timeouts of about 15 minutes in other instances. We have opted for Lambda in this project due to its ease in integration and support for python which we have used to implement our backend logic appropriately.

Dynamo DB was used for storage of incident data. The database handles storage in a consistent way with high availability and fast read and write performance in the range of milliseconds. It allows for seamless integration with Lambda functions by provisioning of predefined functions that can be used to modify the data in the database as need be. Its disadvantage is that query flexibility is limited compared to relational database which allow for normalization. We selected the database because it is able to scale without further manual provisioning making it reliable in fluctuation in load and read and write requests.

Amazon Cognito handles authentication and authorization by use of provisioned credentials. We are able to provision secure authentication out of the box without writing any code. This task would be arduous if we decided to code the logic of the application. It also offers an advantage by its support for federated identities which are easily implemented. Its disadvantage is that there is less customization options for the already designed UI provided and minimal debug and error logging support. Cognito provides the appropriate secure authentication features for our web application with minimal setup hence why it was chosen.

**LIBRARY DESCRIPTION**

The custom library was developed using python to support the functionality provided by the web application. It is used to process incident data within the lambda functions. It ensures a modularized approach of validating the fields required to submit an incident and upon validation storage into our database. The design ensures code maintainability and alignment to software engineering standards by having a separate library validate the code. A snippet of the library code is as shown below:

class IncidentHelper:

    def \_\_init\_\_(self, incident):

        self.incident = incident

    def is\_valid(self):

        required\_fields = ['incidentid', 'title', 'date', 'location', 'description']

        return all(field in self.incident and self.incident[field] for field in required\_fields)

    def normalize(self):

        self.incident['severity']=self.incident.get('severity', 'Low')

        return self.incident

    def summary(self):

        return {

            'incidentid': self.incident['incidentid'],

            'title': self.incident['title'],

            'wordCount': len(self.incident.get('description', '').split())

        }

The library provides functionality such as validation by ensuring all required fields are provided before trying to save the data into a database. It also provides additional functionality such as formatting of the returned incident to be readable.

The library is imported within the handler function and a check of the inputs provided done before saving any incident data to the appropriate database.

**IMPLEMENTATION**

The implementation of the project involved coding of the frontend which involved creation of HTML, CSS and JavaScript files. Three pages were coded which are the home, create and incident pages. The logic was coded using JavaScript to access the backend services provided by AWS.

The frontend files were deployed to Amplify by specifying that we were to source our files from GitHub. In this way, we are able to make changes to our site directly by pushing our code to GitHub.

We begin by creating and deploying our backend services. We create the DynamoDB table programmatically using the following code snippet

import boto3

dynamodb = boto3.client('dynamodb')

# define name and schemas for the table.

def create\_incident\_table():

    try:

        dynamodb.create\_table(

            TableName='Incident',

            KeySchema=[{'AttributeName': 'incidentId', 'KeyType': 'HASH'}],

            AttributeDefinitions=[{'AttributeName': 'incidentId', 'AttributeType': 'S'}],

            BillingMode='PAY\_PER\_REQUEST'

        )

        print("✅ Incidents table created.")

    except dynamodb.exceptions.ResourceInUseException:

        print("⚠️ Table already exists.")

Cognito used for authentication and authorization is also setup using the following code snippet.

import boto3, json

cognito = boto3.client('cognito-idp')

def setup\_user\_pool():

    pool = cognito.create\_user\_pool(PoolName="SiteVisionUserPool")

    pool\_id = pool['UserPool']['Id']

    client = cognito.create\_user\_pool\_client(

        UserPoolId=pool\_id,

        ClientName="WebAppClient",

        GenerateSecret=False,

        ExplicitAuthFlows=["ALLOW\_USER\_PASSWORD\_AUTH", "ALLOW\_REFRESH\_TOKEN\_AUTH"]

    )

    client\_id = client['UserPoolClient']['ClientId']

    print("✅ User Pool ID:", pool\_id)

    print("✅ Client ID:", client\_id)

    return pool\_id, client\_id

setup\_user\_pool()

The user pool is what controls the users who are allowed to access our application and provides us with a link to embed in our login button.

API gateway which integrates lambda functions and appropriate HTTP methods is also created programmatically. We create the API first with the following snippet

api = boto3.client('apigateway')

lambda\_client = boto3.client('lambda')

# 1. Create REST API

api\_resp = api.create\_rest\_api(name="SiteVisionAPI")

api\_id = api\_resp['id']

# 2. Get root resource

root\_id = api.get\_resources(restApiId=api\_id)['items'][0]['id']

# 3. Create /incident resource

resource = api.create\_resource(

    restApiId=api\_id,

    parentId=root\_id,

    pathPart='incident'

)

With the API gateway already in place we can define our routes and specify the lambda functions that will be associated with the routes depending on the type of request to be made. This is shown in the following snippet

def add\_lambda\_route(function\_name, method):

    lambda\_arn = lambda\_client.get\_function(FunctionName=function\_name)['Configuration']['FunctionArn']

    api.put\_method(

        restApiId=api\_id,

        resourceId=res\_id,

        httpMethod=method,

        authorizationType='NONE'

    )

    api.put\_integration(

        restApiId=api\_id,

        resourceId=res\_id,

        httpMethod=method,

        type='AWS\_PROXY',

        integrationHttpMethod='POST',

        uri=f'arn:aws:apigateway:us-east-1:lambda:path/2015-03-31/functions/{lambda\_arn}/invocations'

    )

    lambda\_client.add\_permission(

        FunctionName=function\_name,

        StatementId=f'api-{method}',

        Action='lambda:InvokeFunction',

        Principal='apigateway.amazonaws.com',

        SourceArn=f'arn:aws:execute-api:us-east-1:\*:{api\_id}/\*/{method}/incident'

    )

We already have code for the logic of the lambda functions which modify our backend logic. The functions need to be deployed in order to be able to be triggered from our frontend. The code for deploying the lambda functions is as shown below. The deployed lambda function focuses on getting, creating and deleting incidents.

import boto3, json, zipfile, os

def zip\_lambda(source\_file, zip\_name):

    with zipfile.ZipFile(zip\_name, 'w') as z:

        z.write(source\_file, arcname=os.path.basename(source\_file))

iam = boto3.client('iam')

lambda\_client = boto3.client('lambda')

role = iam.create\_role(

    RoleName='SiteVisionLambdaRole',

    AssumeRolePolicyDocument=json.dumps({

        "Version": "2012-10-17",

        "Statement": [{

            "Effect": "Allow",

            "Principal": {"Service": "lambda.amazonaws.com"},

            "Action": "sts:AssumeRole"

        }]

    })

)

iam.attach\_role\_policy(

    RoleName='SiteVisionLambdaRole',

    PolicyArn='arn:aws:iam::aws:policy/AmazonDynamoDBFullAccess'

)

iam.attach\_role\_policy(

    RoleName='SiteVisionLambdaRole',

    PolicyArn='arn:aws:iam::aws:policy/service-role/AWSLambdaBasicExecutionRole'

)

# Wait for IAM role to propagate before creating Lambda

import time; time.sleep(10)

def deploy\_lambda(name, handler, filename):

    zip\_lambda(filename, f'{name}.zip')

    with open(f'{name}.zip', 'rb') as f:

        return lambda\_client.create\_function(

            FunctionName=name,

            Runtime='python3.9',

            Role=role['Role']['Arn'],

            Handler=handler,

            Code={'ZipFile': f.read()},

            Timeout=15,

            MemorySize=128

        )

deploy\_lambda('CreateIncident', 'insertincident.lambda\_handler', 'lambda/insertincident.py')

deploy\_lambda('GetIncidents', 'getincident.lambda\_handler', 'lambda/getincident.py')

deploy\_lambda('DeleteIncident', 'deleteincident.lambda\_handler', 'lambda/deleteincident.py')

The above deployment is what couples our lambda functions to the respective API gateway for purposes of connecting each API request to the appropriate lambda function trigger to ensure the smooth functionality of our application.

**CI/CD**

Our CI/CD pipeline is handled by Amplify and GitHub. Any changes we make in our code and pushed to GitHub are directly reflected on our site via Amplify. This is a great approach with minimal configuration and ensures that there is ease in deployment of the application. The link to our deployed application is [here](https://main.d2i0tvvol3feh1.amplifyapp.com/).

**CONCLUSION AND REFLECTION**

Site-vision demonstrates the capability of serverless backends as provisioned by cloud computing platforms. It is proof that we can build an end-to-end application with minimal code with most of the features provided out of the box. The use of AWS services boosts development time and reduces overhead costs through features such as scalability and resilience.

If I were to develop the project again, I would include more features and integrate other AWS services geared towards artificial intelligence and machine learning. I would also focus on developing the backend first since it takes up bulk of the work. I would experiment and do more research on which services could support a high number of users and do a side-by-side comparison for effective choice.

The project could also use a good frontend with adoption of modern frameworks to see how they behave when deployed on cloud services. Overall, the project was a success and most of the services selected got the task effectively done. The project led to a great understanding of how cloud native apps are developed and the skills can be extrapolated to other projects.

**REFERENCES**

1. Amazon Web Services (2024) *AWS Lambda – Run code without thinking about servers*. Available at: <https://aws.amazon.com/lambda/> (Accessed: July 2025).
2. Amazon Web Services (2024) *Amazon Cognito – Secure user sign-up, sign-in, and access control*. Available at: <https://aws.amazon.com/cognito/> (Accessed: July 2025).
3. Amazon Web Services (2024) *Amazon API Gateway – Build and deploy APIs*. Available at: <https://aws.amazon.com/api-gateway/> (Accessed: July 2025).
4. Amazon Web Services (2024) *AWS Amplify – Frontend web and mobile development*. Available at: <https://aws.amazon.com/amplify/> (Accessed: 14 July 2025).
5. Amazon Web Services (2024) *Amazon DynamoDB – Fast and flexible NoSQL database service*. Available at: <https://aws.amazon.com/dynamodb/> (Accessed: 14 July 2025).