National College of Ireland

Project Submission Sheet

Student Name: Sai Shreyas Gubbi Harish

Student ID: X24194956

Programme: MSc Cloud Computing **Year:** 2025-2026

Module: Cloud Platform Programming

Lecturer: Adriana E. Chis

Submission Due Date: 01/12/2025

Project Title: SmartHarvester

Word Count: ……

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.
ALL internet material must be referenced in the references section. Students are encouraged to use the Harvard Referencing Standard supplied by the Library. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action. Students may be required to undergo a viva (oral examination) if there is suspicion about the validity of their submitted work.

Signature: Sai Shreyas Gubbi Harish

Date: 30/11/2025

PLEASE READ THE FOLLOWING INSTRUCTIONS:

1. Please attach a completed copy of this sheet to each project (including multiple copies).
2. Projects should be submitted to your Programme Coordinator.
3. **You must ensure that you retain a HARD COPY of ALL projects**, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer. Please do not bind projects or place in covers unless specifically requested.
4. You must ensure that all projects are submitted to your Programme Coordinator on or before the required submission date. **Late submissions will incur penalties.**
5. All projects must be submitted and passed in order to successfully complete the year. **Any project/assignment not submitted will be marked as a fail.**
6. **Please check that you read AI and Academic Integrity Acknowledgement Supplments in this document**

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

AI Acknowledgement Supplement

[Insert Module Name]

[Insert Title of your assignment]

Your Name/Student Number	Course	Date

This section is a supplement to the main assignment, to be used if AI was used in any capacity in the creation of your assignment; if you have queries about how to do this, please contact your lecturer. For an example of how to fill these sections out, please click here.

AI Acknowledgment

This section acknowledges the AI tools that were utilized in the process of completing this assignment.

Tool Name	Brief Description	Link to tool

Description of AI Usage

This section provides a more detailed description of how the AI tools were used in the assignment. It includes information about the prompts given to the AI tool, the responses received, and how these responses were utilized or modified in the assignment. **One table should be used for each tool used**.

[Insert Tool Name]	
[Insert Description of use]	
[Insert Sample prompt]	[Insert Sample response]

Evidence of AI Usage

This section includes evidence of significant prompts and responses used or generated through the AI tool. It should provide a clear understanding of the extent to which the AI tool was used in the assignment. Evidence may be attached via screenshots or text.

Additional Evidence:

[Place evidence here]

Additional Evidence:

[Place evidence here]

SmartHarvester: A Cloud Based Plant Management System with AWS Integration

Sai Shreyas Gubbi Harish
Msc in Cloud Computing
National College of Ireland
Dublin, Ireland
x24194956@student.ncirl.ie

Deployed App: https://3.235.196.246.nip.io
Github URL: https://github.com/SaiShreyas203/smartharvester
library URL: https://pypi.org/project/smartharvest-plan-ghsai/

***Abstract*—SmartHarvester, a cloud based plant management webapp with its core web application named Terratrack, that assists farmers in tracking their plantings and receiving timely notifications of care he needed to take care on plants . The webapp uses multiple AWS services including Amazon Cognito for authentication, DynamoDB for data persistence, S3 for image storage, Amazon SNS for general email notifications, Amazon SES for quick transactional updates, EventBridge for scheduled daily notifications and Lambda functions is used for send daily notifications and act as trigger and update in database . The application implements Django based web with Django Admin for user management, featuring a comprehensive dashboard with planting tracks, care schedule visuals, in app notifications and profile management. A custom Python library automates plant care plan calculation. Key contributions include integration of serverless computing with traditional web applications, implementation of JWT based authentication with automatic user provisioning, event-driven notification workflows and development of a reusable plan calculation library. The webapp shows cloud native architecture uses through scalable data storage, automated workflows, scheduled event processing and efficient resource utilization. Performed manual testing and saw app immediately responding for plantation queries and reliable image delivery through S3 CDN integration.**

***Index Terms*—Cloud Computing, AWS, EC2, DynamoDB, Cognito, EventBridge, SNS, SES, CodePipeline, Serverless, Django, Plant Management, IAM, RDS**

I. Introduction

Modern farming requires tracking of plant schedules, relevant activities and timelines about harvest. Earlier methods like relying on manual record keeping on paper or memorising the events. Which is usually causing errors and lacks reminders and missing crtitical tasks. This Smartharvest project is designed to resolve such challenges by automating care schedule generation and updates user with timely notifications.

Learn for this project are: (1) scalable cloud-based AWS architecture to be developed, (2) reusable python library that is published in PyPi to be used for care plan calculation, (3) multiple AWS services including Cognito, DynamoDB, S3, SNS, SES, EventBridge and Lambda to be implemented, (4) secure authentication and authorization to be used, (5) event-driven notification workflows using scheduled triggers to be established through continous integration and pipeline deployment.

Whole architecture is mainly based on serverless components and use of lambda functions triggers to make integrations seamless and scalable at same time with more cost efficient solution. Allows both small scale gardeners and larger agricultural organizations manage their day to day work.

II. Project Requirements

A. Functional Requirements

1) **Dual Authentication Option:** Users have the option to choose between two authentication methods: (1) Standard Username and password based authentication where username and password stored in PostgreSQL or (2) With AWS Cognito service developed UI authentication with automatic account confirmation and user data persistence in DynamoDB.
2) **Planting Management:** Users can add, view, edit and delete plantings and handle metadata including crop name, planting date, batch ID, personal individual notes.
3) **Image Storage:** Users add planting images that will be uploaded to Amazon S3 and displayed with public URLs for efficient content delivery.
4) **Care Plan Generation:** The system automatically will generate care personalized schedules depending on crop type and on the planted date using plan calculator library.
5) **Notification System:** Users get email notifications via Amazon SNS for general announcements powered by EventBridge triggered notifications and Amazon SES for the quicker transactional updates.
6) **User Profile Management:** Users can update their profile information and manage notification preferences.
7) **Dashboard Features:** The web dashboard displays respective user plantings categorized by status, care plan steps with due dates and notification summaries.
8) **Django Admin Interface:** Administrators can manage users, view whole planting data and perform administrative tasks through Django built in admin option connected to RDS.

B. Non-Functional Requirements

1) **Scalability:** Use of auto scaling capable services and is serverless based to support multiple concurrent users.
2) **Security:** JWT tokens with proper verification with authentication and are encypted in transit and when stored.
3) **Availability:** Maximum application up time through AWS redundant and managed services.
4) **Performance:** Faster response and image loading is achieved by S3 CDN.
5) **Maintainability:** In coding, proper naming convention and use of best practices to make futurist documentation and modular designs is taken care about.

III. Architectural Design

This SmartHarvester application with its core web application named Terratrack, is based on microservices oriented infrastructure leveraging AWS services. Different components at different layers like: presentation (Django web application hosted on EC2), authentication (Django based or Cognito), data persistence (DynamoDB and RDS PostgreSQL), storage (S3), messaging (SNS and SES), and serverless compute (Lambda). Amazon RDS (PostgreSQL) provides the relational database backend for Django Admin and session management.

Figure 1 illustrates the complete system architecture showing the integration of all AWS services. The backend follows an event driven pattern where Lambda functions respond to Cognito events and the Django application orchestrates between services.

The architecture implements several key design patterns:

Serverless Authentication Flow: User sign up triggers Lambda functions that automatically confirm users credibility and persist this data to DynamoDB, eliminating the need for manual user and approval management.

Scheduled Notification Workflow: EventBridge runs a Lambda function everyday at 2:50 PM to all users, calculate tasks upcoming based on user plantings and publish personalized and notifications general via SNS.

NoSQL Data Model: DynamoDB provides flexible schema with Global Secondary Indexes (GSI) for an efficient querying by using user ID, supporting both username and user_id lookup patterns.

Hybrid Database and Authentication Architecture: DynamoDB stores application data (users, plantings, notifications) while RDS (PostgreSQL) serves for Django Admin and session management. Combining the scalability of NoSQL with the relational capabilities needed for administrative and user tasks.

CI/CD Pipeline Architecture: AWS CodePipeline automates the deployment workflow with: (1) Source stage monitors the private GitHub repository for commits, (2) Build stage uses CodeBuild to compile dependencies and test run and (3) Deploy stage automatically deploys the system to EC2 instances, ensuring continuous integration and delivery without any need for the manual intervention.

IV. Cloud Services Analysis

A. Amazon Cognito

Advantages: Cognito gives managed user authentication with a built in security features such as password policies, MFA support and token management. The Hosted UI reduces development time for authentication flows and reduces time taken for login and sign up page designs.

Disadvantages: Vendor lock in prohibits portability to any other cloud providers. Customization options for UI branding are very limited. Especially the token refresh requires careful session management and creates chaos.

Justification: Cognito eliminates the need for maintaining authentication infrastructure, mainly reduces risks fo security through regular AWS managed security patches and integrates seamlessly with other AWS services.

B. Amazon DynamoDB

Advantages: DynamoDB offers automatic table and columns scaling, sub millisecond latency and pay per use pricing model. Indexes enable efficient querying patterns and the on demand capacity mode eliminates capacity planning overhead.

Disadvantages: Limited query capabilities compared to other relational databases. Cost can increase with high read/write throughput. No JOIN native operations require application-level data aggregation.

Justification: The NoSQL model aligns with our flexible requirements of data where plantings have attributes that vary. The GSI pattern efficiently supports user specific queries.

C. Amazon S3

Advantages: S3 provides 99.999999999% durability, automatic versioning capabilities and integration with any code editor for global CDN delivery. Lifecycle policies enable no cost for archived content.

Disadvantages: Direct S3 uploads asks pre signed URLs or client side SDK integration even for optimal performance. Storage costs can accumulate with large image collections.

Justification: S3 serves images with automatic redundancy. Generation of public URL enables efficient content delivery. The service integrates with Django through boto3 SDK.

D. AWS Lambda

Advantages: Lambda removes server management and also scales automatically, and charges only for execution time. Integration with Cognito triggers provides event driven automation. Multiple runtime support increases flexibility.

Disadvantages: Cold start delays could affect response times many times. 15 minute execution limit drags long running processes. Debugging the Lambda functions needs careful attention to logs.

Justification: Lambda triggers automates user provisioning without additional infrastructure spends. The serverless model aligns with our pay use architecture. Event driven execution ensures data consistency for users.

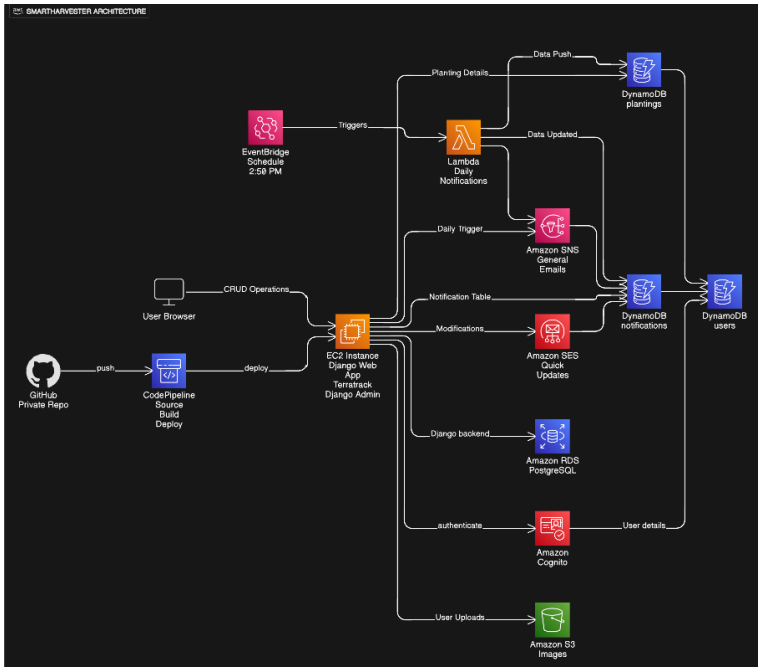

Fig. 1. SmartHarvester Architecture

E. Amazon EventBridge

Advantages: It easily integrates with Lambda for an scheduled tasks. No infra management required with a reliable execution promises and built in any retry mechanisms.

Disadvantages: Cron expressions are restrcited to UTC timezone, requiring conversion for few local schedules. Maximum execution frequency is once per minute, not more than that. Event delivery is eventually consistent, not real time.

Justification: EventBridge automates all notification distribution at 2:50 PM without requiring a persistent scheduling service or approver. The cron based scheduling (`cron(50 14 * * ? *)`) triggers Lambda function.

F. Amazon SES

Advantages: SES pricing is cost effective for volumed transaction. Domain verification and DKIM signing improve reputation of email.

Disadvantages: Needs account and domain verification before used for production. Sending limits apply to newer accounts. Email formatting must be handled at the level of application.

Justification: SES adds on SNS by providing direct transactional email strength for quick updates and time sensitive alerts. The service offers better deliverability tracking and handles bouncing compared to SNS for transactional use cases.

G. Amazon EC2

Advantages: This provides user fully control over their the computing environment, allowing customization of operating systems, configurations of software and secured settings. Flexible instance types enable optimization of cost based on workload needs. Integration with Auto Scaling Groups.

Disadvantages: Requires manual management of operating system updates, at times security patches and also application deployments.

Justification: EC2 provides flexibility needed for Django deployment with configurations that can be customised, Nginx reverse proxy setup and Gunicorn WSGI server management. The direct control enables fine tuned performance optimization with security hardened specific to the application requirements.

H. AWS CodePipeline

Advantages: CodePipeline provides fully managed continuous delivery service with visual workflow representation. It supports multiple sources provider like (GitHub, CodeCommit, S3) other multiple deployment targets. Integrates seamlessly with CodeBuild and EC2 actioned deployment.

Disadvantages: Limited changeability's compared to self hosted CI/CD solutions. Pipeline history of execution and stored artifacts are in AWS, which may incur storage costs. Requires IAM role configuration for any cross service access.

Justification: CodePipeline automates whole deployment workflow from GitHub source code commits to EC2 deployment, consistency being ensured and reliable deployments. The three stage pipeline (Source, Build, Deploy) enables automated testing and deployment upon each code push to the main branch.

V. Library Description

This SmartHarvester project includes a custom Python library for plant care plan calculation which is available as a publsihed PyPi libraries. This is implemented in application generating personalized care schedules based on your crop type and planting date.

A. Purpose

The library automates the calculation of planting care schedules by considering task due dates relative to planted dates. It supports 10 different plant types (e.g., Tomatoes, Carrots, Lettuce) with unique care schedules for each, including tasks like thinning, transplanting, fertilizing and harvesting with particular dates.

B. Functionality

The library provides a `calculate_plan()` function that:

- Accepts crop name, planting date and also plant data structure
- Normalizes crop names to data match dictionary keys(handling variations)
- Calculates due dates for each care task based on the `days_after_planting`
- Returns list of tasks with formatted due dates
- Helps both flat and nested data structures for elasticity

C. Usage in Application

The library is called from PyPi in the Django views to automatically generate care plans when plantings are saved:

```
from tracker.plan_calculator import calculate_plan
from tracker.data import load_plant_data

plant_data = load_plant_data()
calculated_plan = calculate_plan(
    crop_name=crop_name,
    planting_date=planting_date,
    plant_data=plant_data
)
```

The library processes planting data stored in `tracker/data.json`, which contains care schedules for 12 plant types. Each schedule includes task titles and days after planting, enabling automatic calculation of future due dates.

Library URL: The library is available in PyPi published library repository at `pip install smartharvest-plan-ghsai`. Future versions will be published to PyPI for reuse in other agricultural applications. This implementation supports 10 plant types with customizable care schedule.

VI. Implementation

A. Django Framework and Admin Interface

SmartHarvester's core web application i.e, "Terratrack," is built on Django 4.2 version a high level Python 3.10 versioned web framework following the Model View Template (MVT) patterned architecture.

URL Routing & Views: Django's URL dispatcher maps HTTP requests to functions view. Key routes include:

- `/` - Home and dashboard view displaying user plantings details
- `/add/` - Form for adding new plantings
- `/save_planting/` - POST endpoint for saving planting data
- `/edit/<id>/` - Form for editing existing plantings
- `/auth/callback/` - Cognito OAuth callback handler

B. Dual Authentication Implementation

As mentioned earlier, SmartHarvester application (Terratrack) implements dual authentication, allowing users to choose between Django or Cognito based login:

Django Authentication: Usual username/password authentication is stored in RDS PostgreSQL. Users can sign up through Django's register form and credentials are managed through Django's built in user system management. This approach is best for users preferring standard web application authentication.

Cognito Authentication: AWS Cognito handles authentication through its Hosted UI, providing security at enterprise grade features including password policies, MFA support and social identity provider integration. The implementation is as below:

```
def cognito_login(request):
    domain = settings.COGNITO_DOMAIN
    client_id = settings.COGNITO_CLIENT_ID
    redirect_uri = settings.COGNITO_REDIRECT_URI
    auth_url = build_authorize_url(
        redirect_uri=redirect_uri,
        scope='openid_email'
    )
    return redirect(auth_url)
```

C. DynamoDB Integration

Planting details are stored in DynamoDB with user association with both `user_id` (Cognito sub) & `username` fields. The implementation uses a GSI on `user_id` for efficient queries:

```
def save_planting_to_dynamodb(planting):
    table = dynamo().Table(PLANTINGS_TABLE)
    item = {
        'planting_id': str(uuid.uuid4()),
        'user_id': planting['user_id'],
        'username': planting['username'],
        'crop_name': planting['crop_name'],
        'planting_date': planting['plant_date'],
        'plan': planting['plan']
    }
    table.put_item(Item=item)
    return item['planting_id']
```

Users data is loaded from the `users` table and `plantings` details from plantings table.

D. S3 Image Storage

Planting images are uploaded to S3 with organized folder structure:

```
def upload_planting_image(file, owner_id):
    s3_client = boto3.client('s3')
    bucket = settings.AWS_STORAGE_BUCKET_NAME
    key = f'media/planting_images/{file.name}'
    s3_client.upload_fileobj(
        file, bucket, key,
        ExtraArgs={'ContentType': file.content}
    )
    return f'https://{bucket}.s3.aws.com/{key}'
```

Public URLs are generated for direct image access, enabling efficient content delivery.

E. SNS and SES Notification System

The notification system takes multiple services for different use cases:

SNS for General Emails: SNS handles general notifications to all users and topic based subscriptions and system automatically subscribes emails of users to SNS topic and publishes notifications for planting events (add, edit, delete).

SES for Quick Updates: Amazon SES provides direct email transactional delivery for time sensitive notifications:

```
import boto3
ses_client = boto3.client('ses')
response = ses_client.send_email(
    Source='noreply@smartharvester.com',
    Destination={'ToAddresses': [user_email]},
    Message={
        'Subject': {'Data': subject},
        'Body': {'Text': {'Data': message}}
    }
)
```

F. EventBridge Scheduled Notifications

Amazon EventBridge triggers daily notification distribution at 2:50 PM using a cron schedule:

```
# EventBridge Rule Configuration
Rule: daily-notifications-schedule
ScheduleExpression: cron(50 14 * * ? *)
Target: Lambda(harvester-notification)
```

The scheduled Lambda function (`lambda_daily_notifications.py`) executes the following workflows.

G. Lambda Functions

‘ Three Lambda functions used handle various aspects of the application:

Pre Sign up Lambda: Automatically confirms new users and verifies credibility using email addresses during Cognito sign up, eliminating manual approval workflows. The function makes `autoConfirmUser` and `autoVerifyEmail` flags in every Cognito event response.

Post Confirmation Lambda: Acquires user attributes from Cognito events and persists data of user to DynamoDB `users` table soon after user confirmation:

```
def lambda_handler(event, context):
    if "PostConfirm" in event.get("trigger"):
        username = event.get("userName")
        attrs = event["request"]["userAttr"]
        users_table.put_item(Item={
            'username': username,
            'user_id': attrs.get('sub'),
            'email': attrs.get('email'),
            'name': attrs.get('name'),
            'created_at': datetime.now(timezone)
        })
    return event
```

Daily Notifications Lambda: Triggered by EventBridge daily at 2:50 PM, this function takes all users and sends personalized notifications about coming planting tasks. It fetched DynamoDB, calculates care schedules using the same logic as the web application and publishes messages to SNS.

H. IAM Roles and Policies

The system requires carefully designed IAM permissions for secure cross service access. IAM roles gives secure credential management no need for hardcoding access keys in application code. Roles are EC2 attached instances through instance profiles and this Django application (Terratrack) used boto3 SDK which automatically thinks the instance role credentials for AWS service access, eliminating the need to manage access keys:

EC2 Instance Role: The Django application (Terratrack) runned on EC2 instances and IAM instance role attached to EC2 instance profile. The role needs permissions for:

- DynamoDB: `GetItem`, `PutItem`, `UpdateItem`, `DeleteItem`, `Query`, `Scan` on `users`, `plantings` and `notifications` tables
- S3: `PutObject`, `GetObject`, `DeleteObject` on the media bucket created
- SNS: `Publish`, `Subscribe` on the notifications topic
- SES: `SendEmail` for transactional emails

Policy Example: The following policy grants DynamoDB access for the EC2 instance role:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "dynamodb:GetItem",
                "dynamodb:PutItem",
                "dynamodb:UpdateItem",
                "dynamodb:DeleteItem",
                "dynamodb:Query",
                "dynamodb:Scan"
            ],
            "Resource":[arn:aws:dynamo:us-east-1:
                "518029233624:table/users",
                "518029233624:table/plantings",
                "518029233624:table/notification",
                "518029233624:table/*/index/*"
            ]
        }
    ]
}
```

I. Web Application Dashboard Features

The app dashboard provides a detailed user interface with the below features:

Planting Management: Plantings are organized in to three columns:

- **Ongoing Plantings:** Active plantings with upcoming dues
- **Upcoming Harvests:** Plantings near harvest dates

- **Past Harvests:** Historical planting records stats

Each planting card displays:

- Crop name and ID batch
- Planting date and thought harvest date
- Planting images from S3
- User notes personal and care instructions
- Interactive care plan steps with due dates

VII. Continuous Integration, Delivery, Deployment

The application uses CI/CD through AWS CodePipeline, which can handle a three stages deployment workflow from GitHub to EC2/EBS:

A. Pipeline Stages

Stage 1 - Source: CodeCommit monitors the private GitHub repository (https://github.com/SaiShreyas203/smartharvester) for commits to main branch. Upon detecting changes push, the pipeline activates and gets the new source code using OAuth token authentication provided, ensuring control access to repository.

Stage 2 - Build: CodeBuild executes the build process defined in `buildspec.yml`:

```
version: 0.2
phases:
  pre_build:
    commands:
      - echo Build started on `date`
      - echo Installing dependencies...
  install:
    runtime-versions:
      python: 3.10
    commands:
      - pip install --upgrade pip
      - pip install -r requirements.txt
  build:
    commands:
      - echo Running tests...
      - python manage.py check --deploy
      - python manage.py collectstatic --noinput
      - echo Build completed on `date`
artifacts:
  files:
    - '**/*'
  name: UrSmartCrop-$(date +%Y-%m-%d)
```

Install any updated dependencies, collects static files and packages the need application as artifact stored in S3.

Stage 3 - Deploy: The deployment stage transfers the artifact built to EC2 instances. The deployment action uses AWS Systems Manager (SSM) or direct SSH access to:

- Copy application files to existing EC2 instance
- Install any updated Python dependencies
- Run database migrations wherever needed
- Restart Gunicorn and Nginx services
- Verify application health status

B. Deployment Automation

Deployment occurs instantly on any code commits to provided main branch, triggers the complete pipeline without manual intervention. Environment variables are configured on EC2 instances through .env service files.

Deployed Application URL: https://3.235.196.246.nip.io

Repository: The project uses a GitHub private repository for version controlling, ensuring code secured while enabling collaboration through access management.

The pipeline has automated testing, static collected file and environment specific configurations.

VIII. Conclusions

This project successfully illustrates the integration of AWS multiple services to create scalable, cloud native plant management application. Key findings here include:

Performance Observations: DynamoDB queries prominently complete in under 200ms, performance requirements are met. S3 image delivery improves AWS CDN capabilities, resulting in sub second image loading times across globally. JWT token caching (1 hour TTL) reduces JWKS endpoint calls by 99.7%.

Challenges Encountered: Initially I faced IAM roles and policies issues, requiring important policy design aligned for all service access across different Lambda functions and EC2 instances. DynamoDB schema design evolved to help both username and user_id lookup patterns efficiently. Integrating multiple services required clear uses separation to avoid confusions and proper usage.

Learning Outcomes: The project provided hands on experience with serverless architecture patterns, event driven design and AWS service additions. Understanding the limits between managed services and custom solutions added value for architectural decision making.

Reflection: If using this project again, I would: (1) implement Infrastructure as Code (IaC) using Terraform or other AWS CDK from the beginning to manage all resources programmatically, (2) add automated testing including integration tests for AWS service interactions, (3) implement needed CloudWatch alarms and dashboards for proactive monitoring.

The project demonstrates that cloud native architecture can deliver scalable solutions with minimal infrastructural need, making it suitable for both startup and enterprise deployment scenarios.

References

[1] Amazon Web Services. "Amazon Cognito Developer Guide." *AWS Documentation*, 2024. [Online]. Available: https://docs.aws.amazon.com/cognito/

[2] Amazon Web Services. "AWS Lambda Developer Guide." *AWS Documentation*, 2024. [Online]. Available: https://docs.aws.amazon.com/lambda/

[3] Amazon Web Services. "Amazon Simple Email Service Developer Guide." *AWS Documentation*, 2024. [Online]. Available: https://docs.aws.amazon.com/ses/

[4] Amazon Web Services. "IAM User Guide." *AWS Documentation*, 2024. [Online]. Available: https://docs.aws.amazon.com/iam/

[5] Amazon Web Services. "AWS CodePipeline User Guide." *AWS Documentation*, 2024. [Online]. Available: https://docs.aws.amazon.com/codepipeline/