

NAME: ADITI SHARMA

COLLEGE: KIET GROUP OF INSTITUTIONS

Blood Bridge: optimizing Lifesaving resources using AWS Services

 **CONTENTS**

1. Executive Summary	1
1.1 Project Title	1
1.2 Project Description	1
1.3 Core Objectives	2
2. System Scenarios & Workflow	3
2.1 Project Overview	3
2.2 Scenario 1 – Emergency Blood Request	4
2.3 Scenario 2 – Regular Donor Management	5
2.4 Scenario 3 – Blood Bank Inventory Update	6
3. System Architecture & Design	7
3.1 AWS Architecture Diagram	7
3.2 Application Architecture	8
3.3 Database Design (DynamoDB Tables)	9
3.4 Entity Relationship Overview	10
4. Project Workflow	11

4.1 AWS Account Setup	11
4.2 Database Setup	12
4.3 IAM Role Configuration	13
4.4 Backend Development	14
4.5 Frontend Integration	15
4.6 Deployment Workflow	16
5. Project Milestones & Implementation	17
5.1 Milestone 1 – AWS Account Setup	17
5.2 Milestone 2 – DynamoDB Table Creation	18
5.3 Milestone 3 – IAM Role Setup	19
5.4 Milestone 4 – Backend Development	20
5.5 Milestone 5 – EC2 Instance Setup	22
5.6 Milestone 6 – Application Deployment	23
6. Testing & Validation	24
6.1 Functional Testing	24
6.2 Integration Testing	25
6.3 Cloud Service Testing	26
7. Results & Discussion	27
8. Conclusion	28

Project Description:

"BloodBridge" is a comprehensive web-based blood bank management system designed to streamline the process of blood donation and distribution. The project leverages Amazon Web Services (AWS) for robust and scalable infrastructure, utilizing

Amazon DynamoDB for secure and efficient data storage and Amazon EC2 for reliable web hosting. The user-friendly web interface allows individuals to register and log in to their personal accounts, creating a seamless experience for both donors and recipients. Once logged in, users are presented with a dashboard that serves as a central hub for all blood-related activities.

The dashboard prominently features current blood requests, allowing users to view real-time needs in their community. Additionally, registered users can easily submit their own blood requests, specifying blood type, quantity, and urgency. This system not only facilitates quick responses to critical blood needs but also fosters a sense of community engagement in the life-saving act of blood donation. By combining modern cloud technology with an intuitive user interface, "BloodBridge" aims to bridge the gap between blood donors and those in need, ultimately saving lives and improving healthcare outcomes.

Scenarios

Scenario 1: Emergency Blood Request

Sarah, a hospital administrator, logs into Life Link during a critical situation. A patient needs a rare blood type urgently. Using her dashboard, Sarah quickly submits a high-priority blood request, specifying the required blood type and quantity. The system immediately notifies potential donors in the area, significantly reducing the time to find a match and potentially saving the patient's life.

Scenario 2: Regular Donor Management

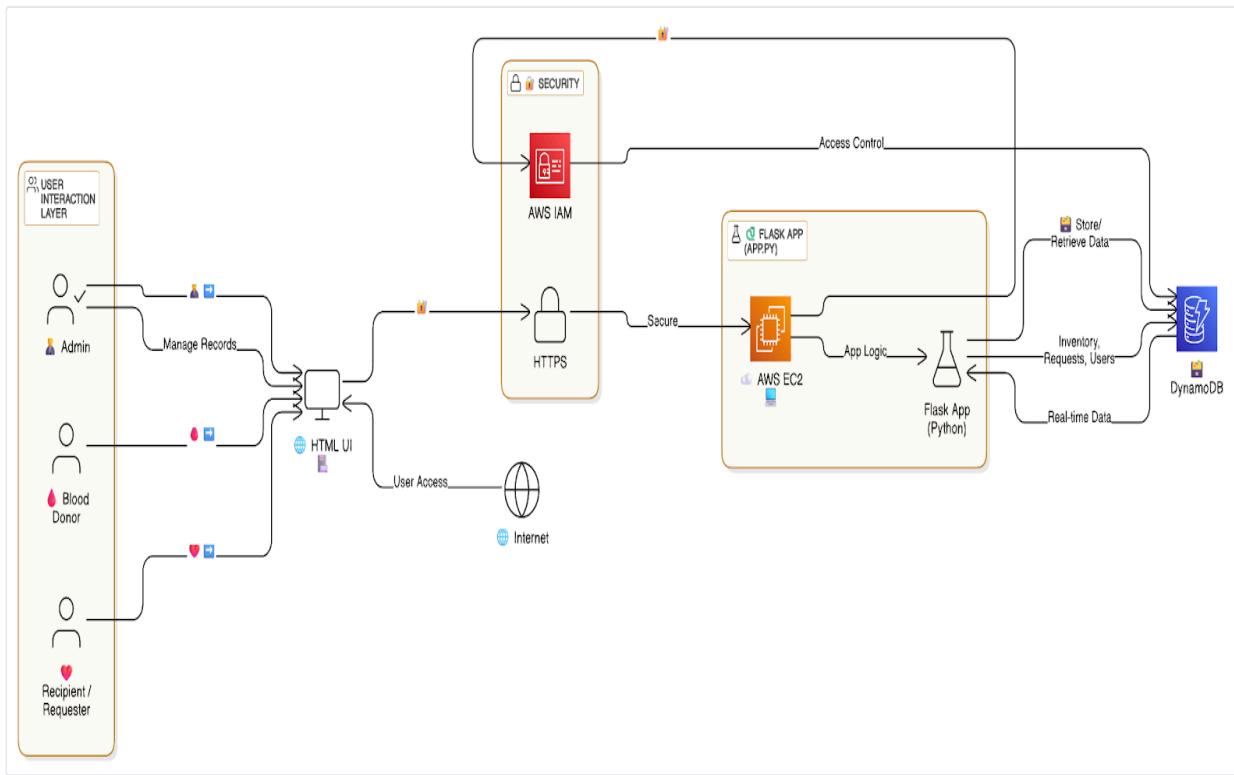
John, a regular blood donor, uses Life Link to manage his donations. After logging in, he checks his dashboard to see when he's eligible to donate again. He notices a nearby blood drive event listed in the requests section. John uses the system to schedule his next donation, helping maintain a steady supply of blood for the local hospitals.

Scenario 3: Blood Bank Inventory Update

A blood bank manager, Lisa, uses Life Link to update the current blood inventory. She logs into her specialized account and accesses a feature to input the latest stock levels for each blood type. The system automatically updates the dashboard for all users, reflecting the current needs. This real-time update helps prioritize requests for blood types that are running low, ensuring efficient distribution of this vital resource.

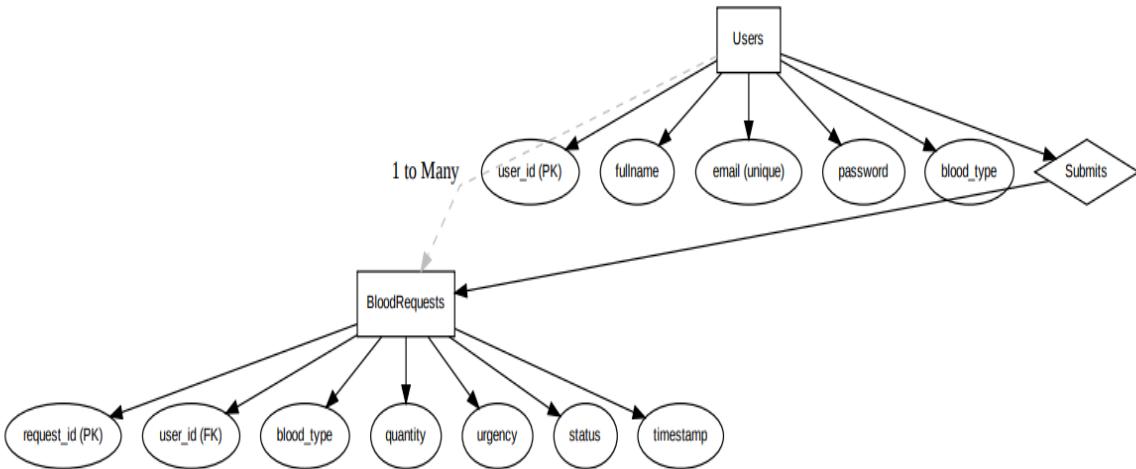
Architecture

This AWS-based architecture powers a scalable and secure web application using Amazon EC2 for hosting the backend, with a lightweight framework like Flask handling core logic. Application data is stored in Amazon DynamoDB, ensuring fast, reliable access, while user access is managed through AWS IAM for secure authentication and control. Real-time alerts and system notifications are enabled via Amazon SNS, enhancing communication and user engagement.



Entity Relationship (ER) Diagram

An ER (Entity-Relationship) diagram visually represents the logical structure of a database by defining entities, their attributes, and the relationships between them. It helps organize data efficiently by illustrating how different components of the system interact and relate. This structured approach supports effective database normalization, data integrity, and simplified query design.



Project Flow

Milestone 1. Backend Development and Application Setup

- Develop the Backend Using Flask.
- Integrate AWS Services Using boto3.

Milestone 2. AWS Account Setup and Login

- Set up an AWS account if not already done.
- Log in to the AWS Management Console

Milestone 3. DynamoDB Database Creation and Setup

- Create a DynamoDB Table.
- Configure Attributes for User Data and Book Requests.

Milestone 4. SNS Notification Setup

- Create SNS topics for book request notifications.
- Subscribe users and library staff to SNS email notifications.

Milestone 5. IAM Role Setup

- Create IAM Role
- Attach Policies

Milestone 6. EC2 Instance Setup

- Launch an EC2 instance to host the Flask application.
- Configure security groups for HTTP, and SSH access.

Milestone 7. Deployment on EC2

- Upload Flask Files

- Run the Flask App

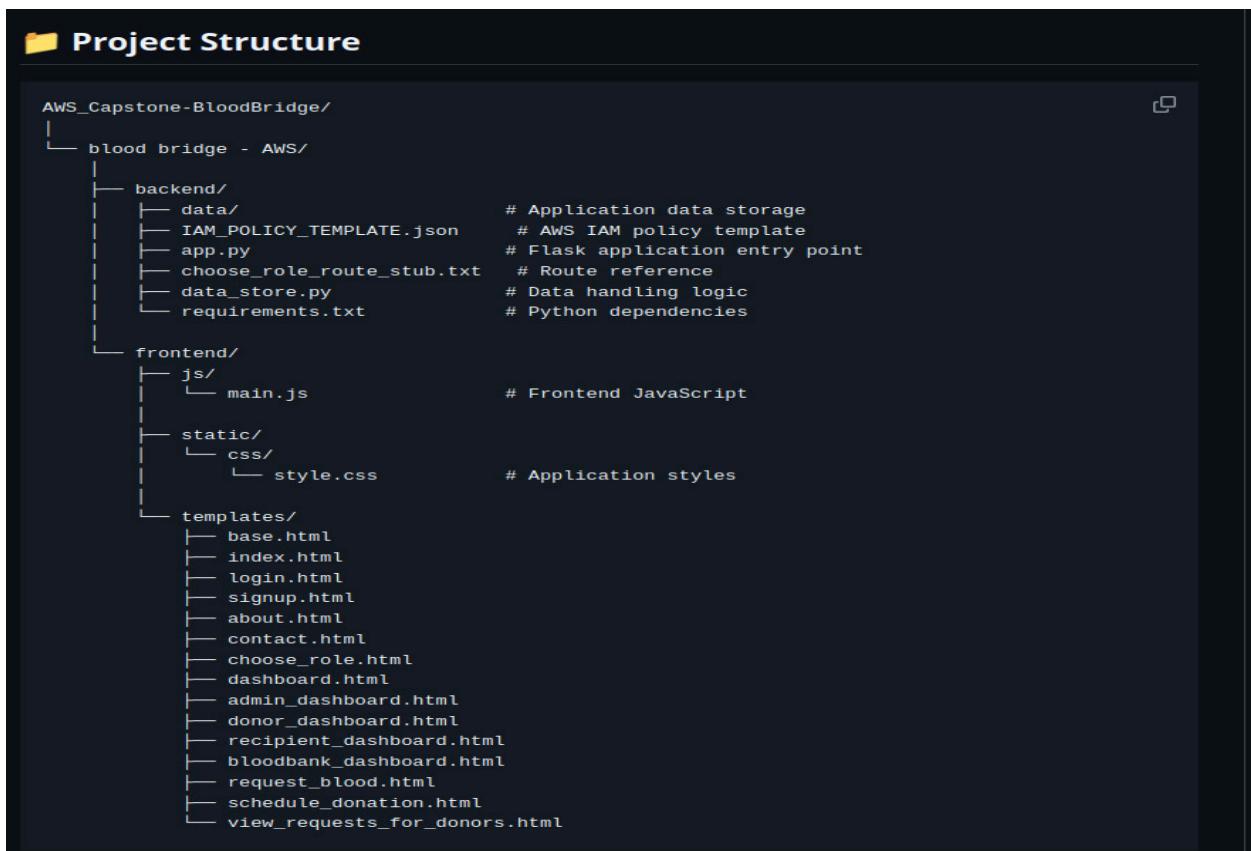
Milestone 8. Testing and Deployment

- Conduct functional testing to verify user signup, login, buy/sell stocks and notifications.

Milestone 1: Web Application Development And Setup

Backend Development and Application Setup focuses on establishing the core structure of the application. This includes configuring the backend framework, setting up routing, and integrating database connectivity. It lays the groundwork for handling user interactions, data management, and secure access.

- Activity 4.1: Develop the backend using Flask.



```
AWS_Capstone-BloodBridge/
└── blood_bridge - AWS/
    ├── backend/
    │   ├── data/
    │   ├── IAM_POLICY_TEMPLATE.json      # Application data storage
    │   ├── app.py                      # AWS IAM policy template
    │   ├── choose_role_route_stub.txt   # Flask application entry point
    │   ├── data_store.py               # Route reference
    │   └── requirements.txt            # Data handling logic
                                    # Python dependencies

    └── frontend/
        ├── js/
        │   └── main.js                  # Frontend JavaScript
        ├── static/
        │   └── css/
        │       └── style.css           # Application styles
        └── templates/
            ├── base.html
            ├── index.html
            ├── login.html
            ├── signup.html
            ├── about.html
            ├── contact.html
            ├── choose_role.html
            ├── dashboard.html
            ├── admin_dashboard.html
            ├── donor_dashboard.html
            ├── recipient_dashboard.html
            ├── bloodbank_dashboard.html
            ├── request_blood.html
            ├── schedule_donation.html
            └── view_requests_for_donors.html
```

Description of the Code Flask App Initialization

● Imports and Configuration

- The application starts by importing necessary modules such as Flask, boto3, session handling, and other utilities like datetime.
- app = Flask(__name__): Initializes a Flask web application.
- app.secret_key = "your_secret_key": This key is necessary for securely handling sessions and flash messages (temporary notifications).

```
1  import logging
2  import os
3  import uuid
4  from datetime import datetime
5
6  import boto3
7  # CloudWatch removed: watchtower import was here.
8  from flask import Flask, render_template, request, redirect, url_for, session, flash
9  from werkzeug.security import generate_password_hash, check_password_hash
10
11 v  app = Flask(
12      __name__,
13      template_folder="../frontend/templates",
14      static_folder="../frontend/static",
15  )
16
17  app.secret_key = os.environ.get("FLASK_SECRET_KEY", "blood_bridge_secret_key")
18
19  # ----- AWS CONFIGURATION -----
20  # IAM ROLE-BASED AUTHENTICATION:
21  # This application uses EC2 Instance IAM Role for AWS authentication.
22  # NO hardcoded AWS credentials (access keys/secrets) are used.
23  #
24  # CREDENTIAL HANDLING:
25  # - boto3 automatically uses the EC2 instance's IAM role credentials via
26  #   the default credential chain (EC2 instance metadata service)
27  # - No explicit credentials are passed to boto3 clients/resources
28  # - When running on EC2, the instance must have an IAM role attached
29  #
30  # REQUIRED IAM ROLE PERMISSIONS:
31  # The EC2 instance IAM role must have the following permissions:
```

Description: The section shows a Flask app (`dynamo_app.py`) that connects to AWS DynamoDB using `boto3`. It sets up a secret key, defines the AWS region, accesses "`users`" and "`requests`" tables, and renders `index.html` at the root route.

```

  AWS_Capstone-BloodBridge / blood bridge - AWS / backend / app.py
  Code Blame 758 lines (614 loc) · 26.9 KB
  11 app = Flask(
  12
  13     AWS_REGION = os.environ.get("AWS_REGION", "us-east-1")
  14
  15     # AWS PERMISSION REQUIRED: DynamoDB read/write access
  16     # boto3 will automatically use EC2 IAM role credentials (no explicit credentials passed)
  17     dynamodb = boto3.resource("dynamodb", region_name=AWS_REGION)
  18     users_table = dynamodb.Table(os.environ.get("USERS_TABLE", "BloodBridgeUsers"))
  19     donations_table = dynamodb.Table(
  20         os.environ.get("DONATIONS_TABLE", "BloodBridgeDonations")
  21     )
  22     requests_table = dynamodb.Table(
  23         os.environ.get("REQUESTS_TABLE", "BloodBridgeRequests")
  24     )
  25     messages_table = dynamodb.Table(
  26         os.environ.get("MESSAGES_TABLE", "BloodBridgeMessages")
  27     )
  28
  29     # AWS PERMISSION REQUIRED: SNS Publish access
  30     # boto3 will automatically use EC2 IAM role credentials (no explicit credentials passed)
  31     sns_client = boto3.client("sns", region_name=AWS_REGION)
  32     SNS_TOPIC_ARN = os.environ.get("BLOOD_REQUEST_SNS_TOPIC_ARN")
  33
  34     # ----- LOGGING CONFIGURATION -----
  35     # CloudWatch logging removed - now using standard Python logging to stdout/stderr
  36     # Logs can be captured by EC2 Cloudwatch agent or container logging if needed
  37     logger = logging.getLogger("bloodbridge")
  38     logger.setLevel(logging.INFO)
  39     logger.addHandler(logging.StreamHandler())
  40
  41     # If not logger.handlers:
  42     #     Stream to stdout (useful for local/dev and when EC2 logging agent is used)
  43     #     console_handler = logging.StreamHandler()
  44     #     console_handler.setLevel(logging.INFO)
  45     #     logger.addHandler(console_handler)
  46     #     # CloudWatch direct logging removed - watchtower handler was here
  47
  48     # ----- DynamoDB HELPERS -----
  49
  50     def _scan_all(table, filter_expression=None, projection_expression=None):
  51         ...

```

Sign Up (/signup):

- In this route the user submits a registration form with details like fullname, email, password, and blood_type.
- It checks if the email already exists in the database.
- If the email exists, the user is redirected to the login page, and a flash message is shown.
- If the user is new, their details are inserted into the database.

```

226
227     @app.route('/signup', methods=['GET', 'POST'])
228     def signup():
229         if request.method == 'POST':
230             account_type = request.form.get('account_type') # '' or 'bloodbank'
231             hashed_password = generate_password_hash(request.form['password'])
232
233             # Generate a stable string ID for DynamoDB; we expose it to templates as _id
234             user_id = str(uuid.uuid4())
235
236             user_doc = {
237                 "user_id": user_id,
238                 "_id": user_id, # keep legacy attribute name for templates
239                 "name": request.form["name"],
240                 "email": request.form["email"],
241                 "password": hashed_password,
242                 # no role stored at signup for normal users
243                 "blood_group": request.form.get("blood_group"),
244             }
245
246             # If account type is bloodbank, assign special role
247             if account_type == "bloodbank":
248                 user_doc["role"] = "bloodbank"
249
250             # Persist user in DynamoDB
251             put_user(user_doc)
252             logger.info("New user signed up", extra={"email": user_doc["email"], "role": user_doc.get("role")})
253
254             flash("Registration successful!")
255             return redirect(url_for("login"))
256
257     return render_template('signup.html')
258
259
260     @app.route('/login', methods=['GET', 'POST'])
261     def login():
262         if request.method == 'POST':
263             user = get_user_by_email(request.form["email"])
264
265             if user and check_password_hash(user["password"], request.form["password"]):
266                 # Normalise ID fields for backwards compatibility

```

User Account Confirmation (/confirm):

- The `/confirm` route retrieves the user from the session and renders a confirmation page

User Login (/login):

- A login form accepts the email and password.
- These credentials are verified by querying the database. If they match, the user is redirected to the dashboard, and session data is stored:

```

259
260     @app.route('/login', methods=['GET', 'POST'])
261     def login():
262         if request.method == 'POST':
263             user = get_user_by_email(request.form["email"])
264
265             if user and check_password_hash(user["password"], request.form["password"]):
266                 # Normalise ID fields for backwards compatibility
267                 uid = user.get("user_id") or user.get("_id")
268                 session["user_id"] = uid
269                 session["name"] = user["name"]
270                 session["user_email"] = user.get("email")
271                 # Preserve any special roles (admin / bloodbank) if they exist
272                 session["role"] = user.get("role")
273
274                 logger.info(
275                     "User logged in",
276                     extra={"user_id": uid, "email": user.get("email"), "role": user.get("role")},
277                 )
278
279                 # Normal users: go to role selection page
280                 if user.get("role") in ["admin", "bloodbank"]:
281                     return redirect(url_for("dashboard"))
282                 else:
283                     return redirect(url_for("choose_role"))
284
285                 flash("Invalid email or password")
286
287             return render_template('login.html')
288
289

```

Description: This block runs the Flask app on the specified host (0.0.0.0) and port (80) in debug mode, allowing for real-time error tracking and hot reloading during development. It ensures the application is accessible from any network interface on the machine.

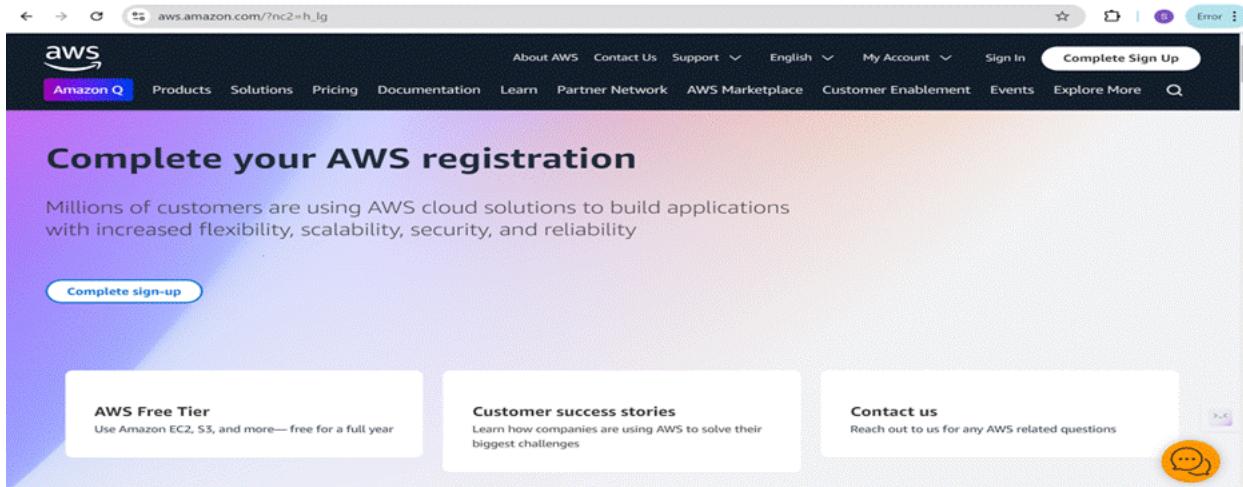
```

745
746
747     if __name__ == '__main__':
748         # Bind to 0.0.0.0 so the app is reachable on EC2.
749         # In production, disable debug and run behind a WSGI server like gunicorn.
750         app.run(host='0.0.0.0', port=int(os.environ.get("PORT", 5000)), debug=bool(os.environ.get("FLASK_DEBUG", False)), use_reloader=False)

```

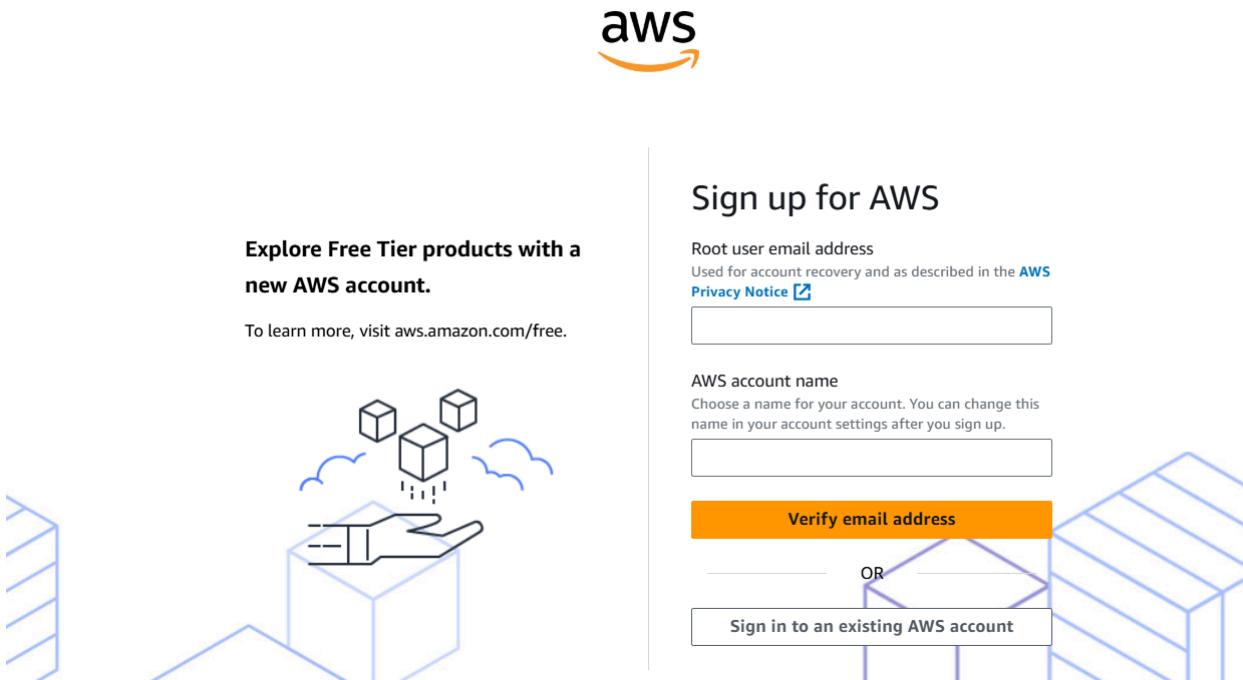
Milestone 2 : AWS Account Setup

An AWS account was created by registering on the official AWS website. During the setup, required details were provided and billing information was configured to enable access to AWS services under the Free Tier.



Activity 1.2: Log in to the AWS Management Console

After successful account creation, the AWS Management Console was accessed using the registered credentials. The console serves as the central platform for managing AWS services used in the Blood Ridge.

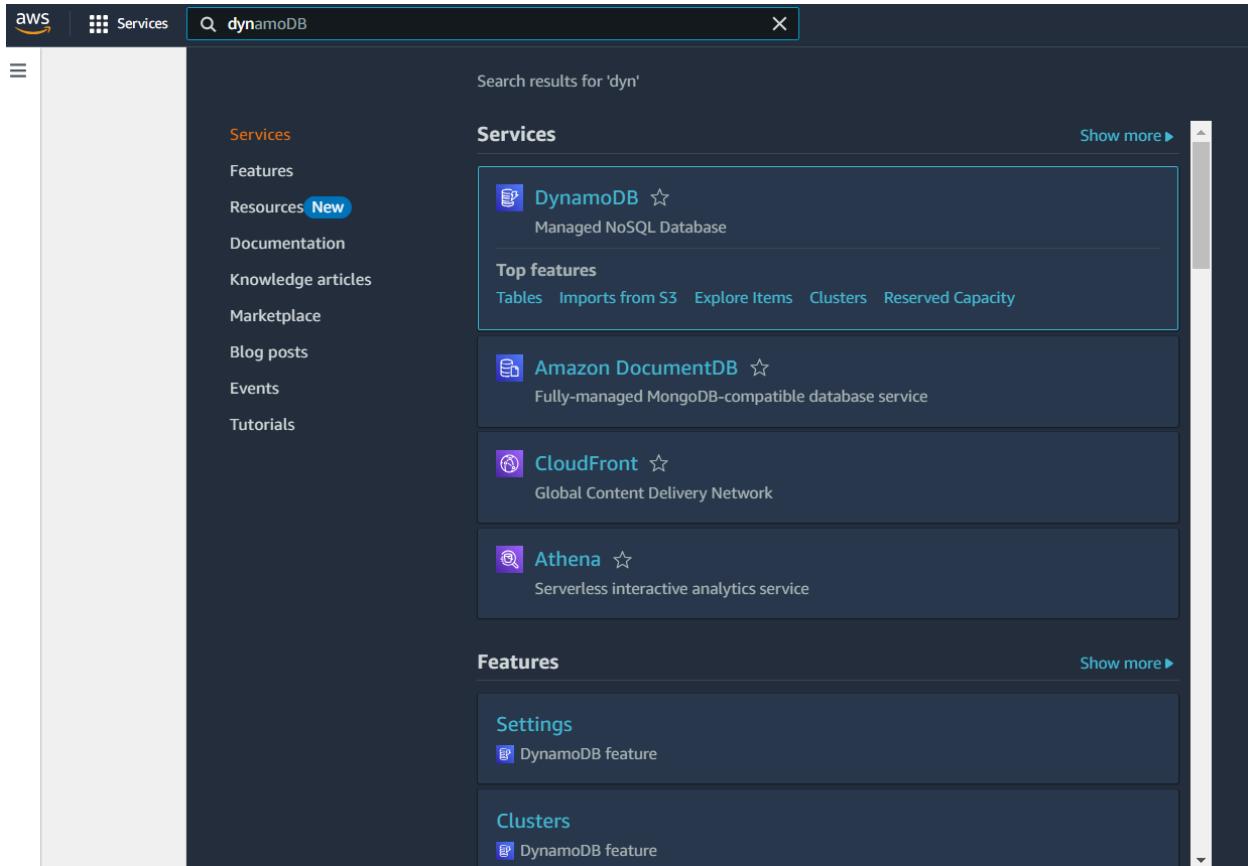


Milestone 3 : DynamoDB Database Creation and

Setup

Database Creation and Setup involves initializing a cloud-based NoSQL database to store and manage application data efficiently. This step includes defining tables, setting primary keys, and configuring read/write capacities. It ensures scalable, high-performance data storage for seamless backend operations.

Navigate to the DynamoDB



The image shows two screenshots of the Amazon DynamoDB console. The top screenshot is the 'Dashboard' page, which includes sections for 'Alarms (0)', 'DAX clusters (0)', and a 'Create resources' panel. The bottom screenshot is the 'Tables' page, showing a table with columns for Name, Status, Partition key, Sort key, Indexes, Deletion protection, Read capacity mode, Write capacity mode, and Total size. It displays a message: 'You have no tables in this account in this AWS Region.' Both screenshots show a sidebar with navigation links like 'Dashboard', 'Tables', 'Explore items', 'PartiQL editor', 'Backups', 'Exports to S3', 'Imports from S3', 'Integrations', and 'Settings'.

Create a DynamoDB table for storing registration details and requests.

The image shows the 'Create table' wizard in the Amazon DynamoDB console. The first step, 'Table details', is shown. It requires a 'Table name' (e.g., 'users') and a 'Partition key' (e.g., 'email'). An optional 'Sort key - optional' field is also present. The second step, 'Table settings', is partially visible at the bottom. The browser's address bar shows the URL: <https://us-east-1.console.aws.amazon.com/dynamodbv2/home?region=us-east-1#create-table>.

The screenshot shows the AWS DynamoDB console interface. On the left, a sidebar menu includes 'Dashboard', 'Tables' (selected), 'Explore items', 'PartiQL editor', 'Backups', 'Exports to S3', 'Imports from S3', 'Integrations', 'Reserved capacity', and 'Settings'. Below this is a 'DAX' section with 'Clusters', 'Subnet groups', 'Parameter groups', and 'Events'. The main content area displays a table titled 'Tables (1) Info'. A blue banner at the top says 'Creating the users table. It will be available for use shortly.' The table has columns: Name, Status, Partition key, Sort key, Indexes, Replication Regions, Deletion protection, Favorite, Read capacity mode, Write capacity mode, and Total size. One row is shown for the 'users' table, which is currently 'Creating' with an email partition key. At the bottom right of the table area, there are 'Actions', 'Delete', and 'Create table' buttons. The status bar at the bottom indicates 'Feb 6 19:20'.

This screenshot shows the AWS DynamoDB console after the 'users' table has been successfully created. The sidebar and table structure are identical to the first screenshot, but the table count has increased to 'Tables (4) Info'. The table list now includes 'blood_requests', 'donations', 'inventory', and 'users', all in an 'Active' state. The status bar at the bottom indicates 'February 6, 2026, 19:24 (UTC+5:30)'.

Milestone 4 : IAM Role Setup

Create IAM Role

- In the AWS Console, go to IAM and create a new IAM Role for EC2 to interact with

DynamoDB and SNS.

The screenshot shows the AWS Lambda search results page. The search bar at the top contains the text 'Iam'. Below the search bar, there is a sidebar with links to 'Services', 'Features', 'Resources New', 'Documentation', 'Knowledge articles', 'Marketplace', 'Blog posts', 'Events', and 'Tutorials'. The main content area is titled 'Search results for 'Iam'' and lists four services: 'IAM' (Manage access to AWS resources), 'IAM Identity Center' (Manage workforce user access to multiple AWS accounts and cloud applications), 'Resource Access Manager' (Share AWS resources with other accounts or AWS Organizations), and 'AWS App Mesh' (Easily monitor and control microservices). Each service entry includes a star icon indicating it is a popular choice.

The screenshot shows the 'Create role' wizard in the AWS IAM console, specifically Step 3: 'Name, review, and create'. The title bar indicates the URL is https://us-east-1.console.aws.amazon.com/iam/home?region=us-east-1#/roles/create?trustedEntityType=AWS_SERVICE&selectedService=EC2&policies=arn%3aaws%3A.... The page shows the 'Role details' section where the 'Role name' is set to 'StudentUser'. The 'Description' field contains the text 'Allows EC2 instances to call AWS services on your behalf.'. Below this, the 'Step 1: Select trusted entities' section shows a trust policy with the following JSON code:

```
1+ [ { 2+   "Version": "2012-10-17", 3+     "Statement": [ 4+       { 5+         "Effect": "Allow", 6+         "Action": [ 7+           "sts:AssumeRole" ] } ] } ]
```

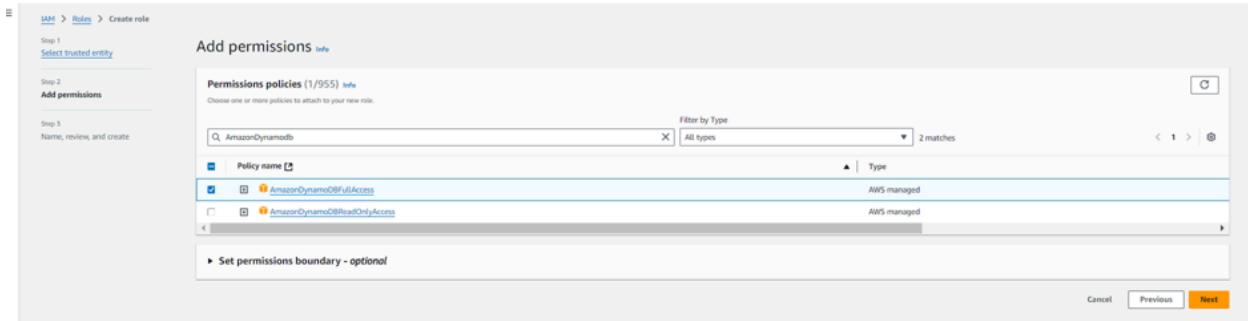
Attach Policies

Attach the following policies to the role:

- AmazonDynamoDBFullAccess: Allows EC2 to perform read/write operations on

DynamoDB.

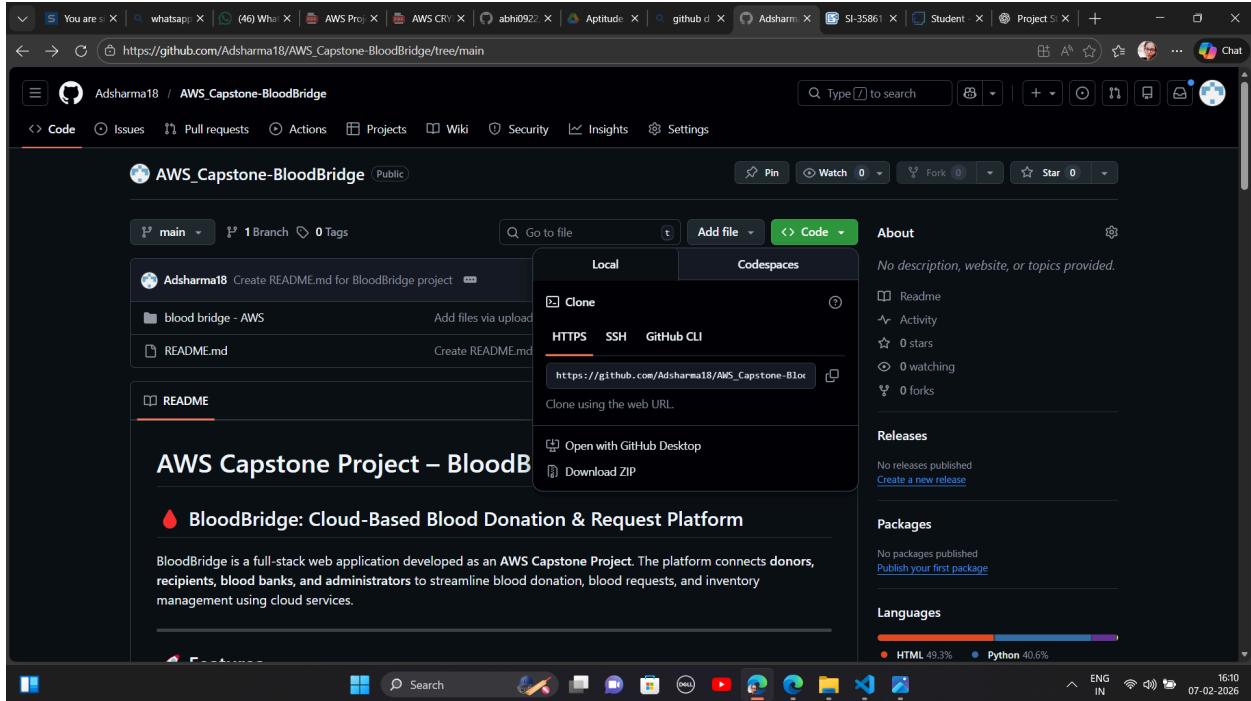
- **AmazonSNSFullAccess:** Grants EC2 the ability to send notifications via SNS

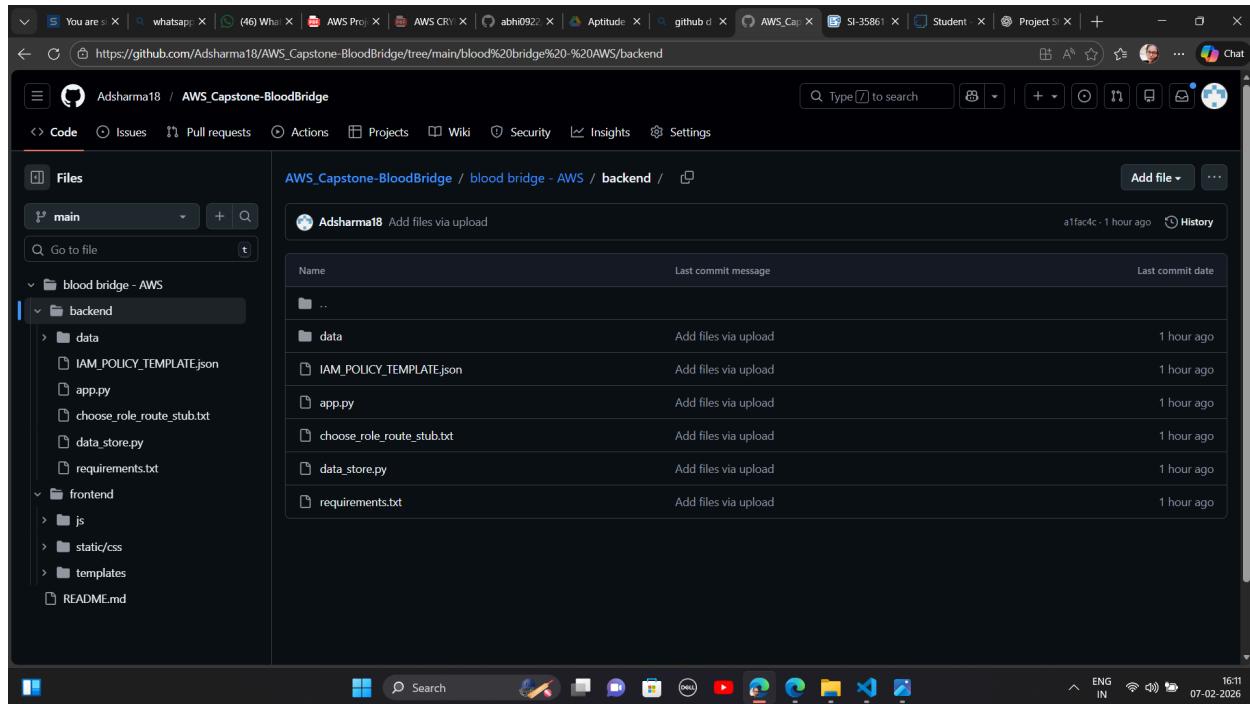


Milestone 5 : EC2 instance Setup

To set up a public EC2 instance, choose an appropriate Amazon Machine Image (AMI) and instance type. Ensure the security group allows inbound traffic on necessary ports (e.g., HTTP/HTTPS for web applications). After launching the instance, associate it with an Elastic IP for consistent public access, and configure your application or services to be publicly accessible.

Milestone 6 : Deployment using EC2

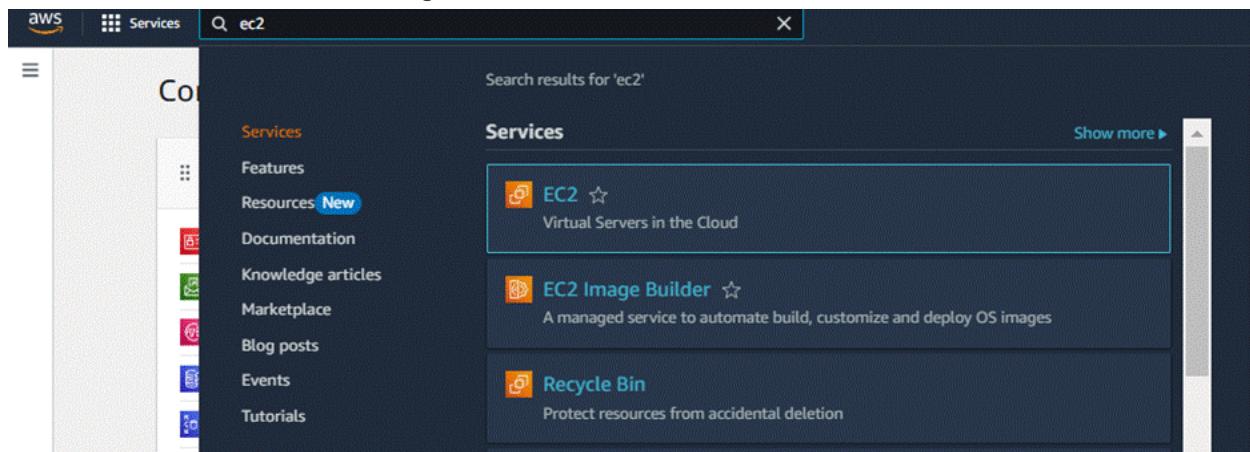




Launch an EC2 instance to host the Flask application.

Launch EC2 Instance

- In the AWS Console, navigate to EC2 and launch a new instance.



Resources

[EC2 Global View](#)   

You are using the following Amazon EC2 resources in the United States (N. Virginia) Region:

Instances (running)	0	Auto Scaling Groups	0	Capacity Reservations	0
Dedicated Hosts	0	Elastic IPs	0	Instances	0
Key pairs	0	Load balancers	0	Placement groups	0
Security groups	0	Snapshots	0	Volumes	0

Launch instance

To get started, launch an Amazon EC2 instance, which is a virtual server in the cloud.

[Launch instance](#) 

[Migrate a server](#) 

Note: Your instances will launch in the United States (N. Virginia) Region

Instance alarms

[View in CloudWatch](#) 

 0 in alarm

 0 OK

 0 insufficient data

Service health

[AWS Health Dashboard](#)  

Region

United States (N. Virginia)

Status

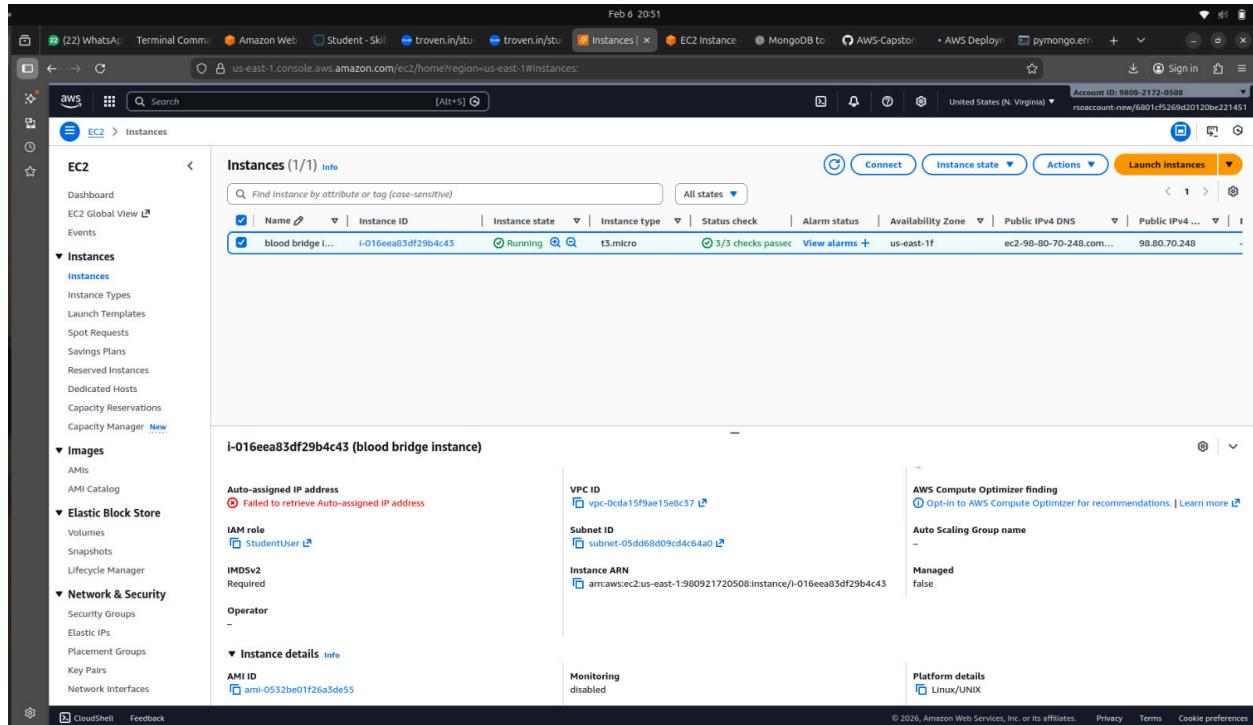
 This service is operating normally.

Zones

Zone name	Zone ID
-----------	---------

us-east-1a	use1-az1
------------	----------

- This is the EC2 instance console page where you need to click on the launch instance to create a new instance or click on the instances to go with the running instances.



Configure security groups for HTTP, and SSH access.

- Select the no key pair for the authentication process and proceed to the networking settings
- In the networking settings select the VPC you have created for your project if not select the default VPC(make sure to stop the services after using them, VPC also will charge you based upon the As per **New – AWS Public IPv4 Address Charge + Public IP Insight**, you'll get billed \$0.005/hour = 0.12/day.)
- Edit the VPC security settings, edit the security configurations of the ec2 which you need to define to allow the incoming and outgoing traffic from your website.

Connect to instance Info

Connect to your instance i-001861022fbcac290 (InstantLibraryApp) using any of these options

[EC2 Instance Connect](#)

Session Manager

SSH client

EC2 serial console



Port 22 (SSH) is open to all IPv4 addresses

Port 22 (SSH) is currently open to all IPv4 addresses, indicated by **0.0.0.0/0** in the inbound rule in [your security group](#). For increased security, consider restricting access to only the EC2 Instance Connect service IP addresses for your Region: 13.233.177.0/29. [Learn more](#).

Instance ID

i-001861022fbcac290 (InstantLibraryApp)

Connection Type

Connect using EC2 Instance Connect

Connect using the EC2 Instance Connect browser-based client, with a public IPv4 or IPv6 address.

Connect using EC2 Instance Connect Endpoint

Connect using the EC2 Instance Connect browser-based client, with a private IPv4 address and a VPC endpoint.

Public IPv4 address

13.200.229.59

IPv6 address

—

Username

Enter the username defined in the AMI used to launch the instance. If you didn't define a custom username, use the default username, ec2-user.

ec2-user X

i Note: In most cases, the default username, ec2-user, is correct. However, read your AMI usage instructions to check if the AMI owner has changed the default AMI username.

[Cancel](#)

[Connect](#)

The screenshot shows a terminal window titled "Amazon Linux 2023" with the URL "https://aws.amazon.com/linux/amazon-linux-2023". The terminal prompt is "[ec2-user@ip-172-31-7-0 ~]#". At the bottom of the terminal, there is a status bar with the text "i-0c6af406cab0f1751 (public-transport)" and "PublicIPs: 3.141.199.115 PrivateIPs: 172.31.7.0".

Milestone 6 : Deployment using EC2

Deployment on an EC2 instance involves launching a server, configuring security groups for public access, and uploading your application files. After setting up necessary dependencies and environment variables, start your application and ensure it's running on the correct port. Finally, bind your domain or use the public IP to make the application accessible online.

Install Software on the EC2 Instance

Install Python3, Flask, and Git:

- **On Amazon Linux 2:**

```
sudo yum update -y  
sudo yum install python3 git  
sudo pip3 install flask boto3
```

- **Verify Installations:**

```
flask --version  
git --version
```

Clone your project repository from GitHub into the EC2 instance using Git.

- Run: 'git clone https://github.com/Adsharma18/AWS_Capstone-BloodBridge.git'
Note: change your-github-username and your-repository-name with your credentials
- here: 'git clone https://github.com/suryavarma03/BloodBridge.git'

```

* Running on http://127.0.0.1:5000
Press CTRL+C to quit
^C(venv) [ec2-user@ip-172-31-72-48 backend]$ git pull
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (4/4), 1.01 KiB | 517.00 KiB/s, done.
From https://github.com/Adsharma18/AWS_Capstone_BloodBridge
   390e391..4f82429 main      -> origin/main
Updating 390e391..4f82429
Fast-forward
 backend/app.py | 2 ++
 1 file changed, 1 insertion(+), 1 deletion(-)
(venv) [ec2-user@ip-172-31-72-48 backend]$ python3 app.py
 * Serving Flask app 'app'
 * Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://172.31.72.48:5000
Press CTRL+C to quit
59.145.191.138 - - [06/Feb/2026 14:52:42] "GET / HTTP/1.1" 200 -
59.145.191.138 - - [06/Feb/2026 14:52:43] "GET /static/css/style.css HTTP/1.1" 200 -
59.145.191.138 - - [06/Feb/2026 14:52:43] "GET /favicon.ico HTTP/1.1" 404 -
59.145.191.138 - - [06/Feb/2026 14:53:19] "GET /signup HTTP/1.1" 200 -
59.145.191.138 - - [06/Feb/2026 14:53:19] "GET /static/css/style.css HTTP/1.1" 304 -
59.145.191.138 - - [06/Feb/2026 14:54:17] "POST /signup HTTP/1.1" 500 -

```

i-05147c06f072d1936 (blood bridge instance)

PublicIPs: 3.238.126.76 PrivateIPs: 172.31.72.48

This will download your project to the EC2 instance.

- To navigate to the project directory, run the following command: cd <your files path>
- Once inside the project directory, configure and run the Flask application by executing the following command with elevated privileges:
- Run the Flask Application:
sudo flask run --host=0.0.0.0 --port=80

Verify the Flask app is running:

<http://your-ec2-public-ip>

- Run the Flask app on the EC2 instance

```

Available Versions:
Version 2023.10.20260202:
Run the following command to upgrade to 2023.10.20260202:
dnf upgrade --releasever=2023.10.20260202

Release notes:
https://docs.aws.amazon.com/linux/al2023/release-notes/relnotes-2023.10.20260202.html

=====
Installed:
git-2.50.1-1.amzn2023.0.1.x86_64          git-core-2.50.1-1.amzn2023.0.1.x86_64
libcrypt-compat-4.4.33-7.amzn2023.x86_64    perl-Error-1.0.17029-5.amzn2023.0.2.noarch
perl-Git-2.50.1-1.amzn2023.0.1.noarch       perl-TermReadKey-2.38-9.amzn2023.0.2.x86_64
python3-pip-21.3.1-2.amzn2023.0.15.noarch    perl-lib-0.65-477.amzn2023.0.7.x86_64

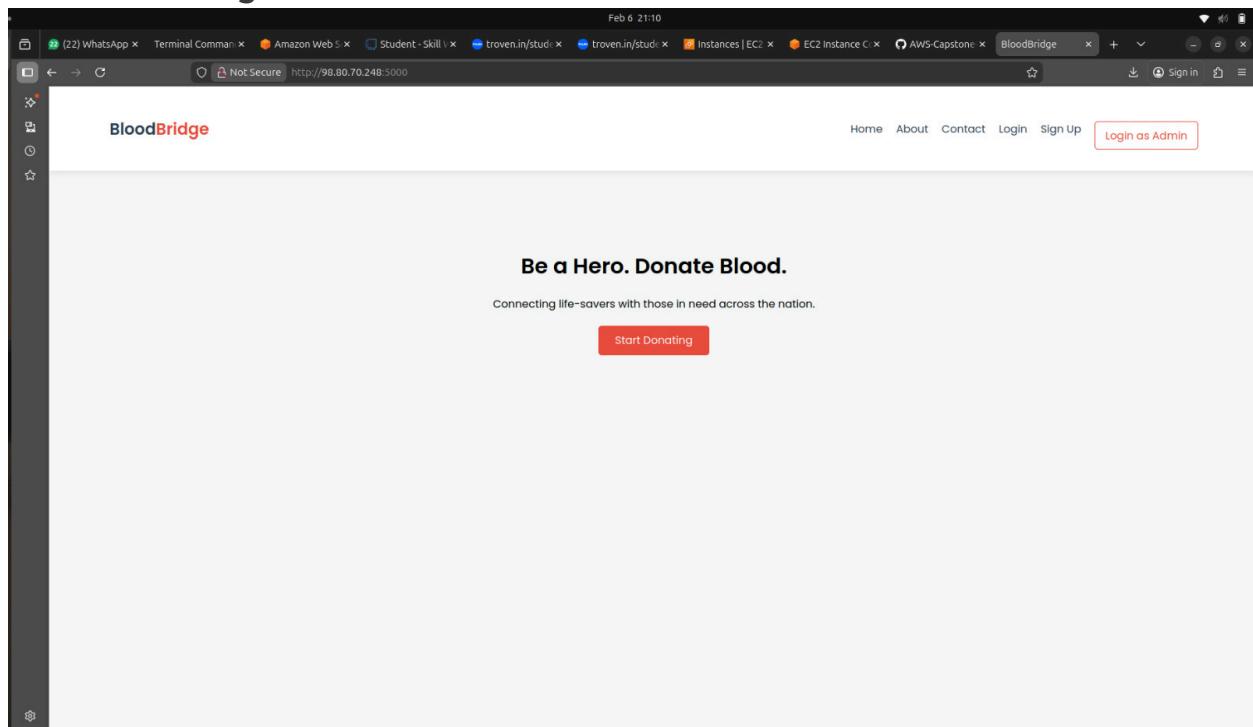
Complete!
[ec2-user@ip-172-31-72-48 ~]$ git clone https://github.com/Adsharma18/AWS_Capstone_BloodBridge.git
Cloning into 'AWS_Capstone_BloodBridge'...
remote: Enumerating objects: 95, done.
remote: Counting objects: 100% (95/95), done.
remote: Compressing objects: 100% (69/69), done.
remote: Total 95 (delta 19), reused 95 (delta 19), pack-reused 0 (from 0)
Receiving objects: 100% (95/95), 54.17 KiB | 6.77 MiB/s, done.
Resolving deltas: 100% (19/19), done.
[ec2-user@ip-172-31-72-48 ~]$ [i-05147c06f072d1936 (blood bridge instance)

PublicIPs: 3.238.126.76 PrivateIPs: 172.31.72.48

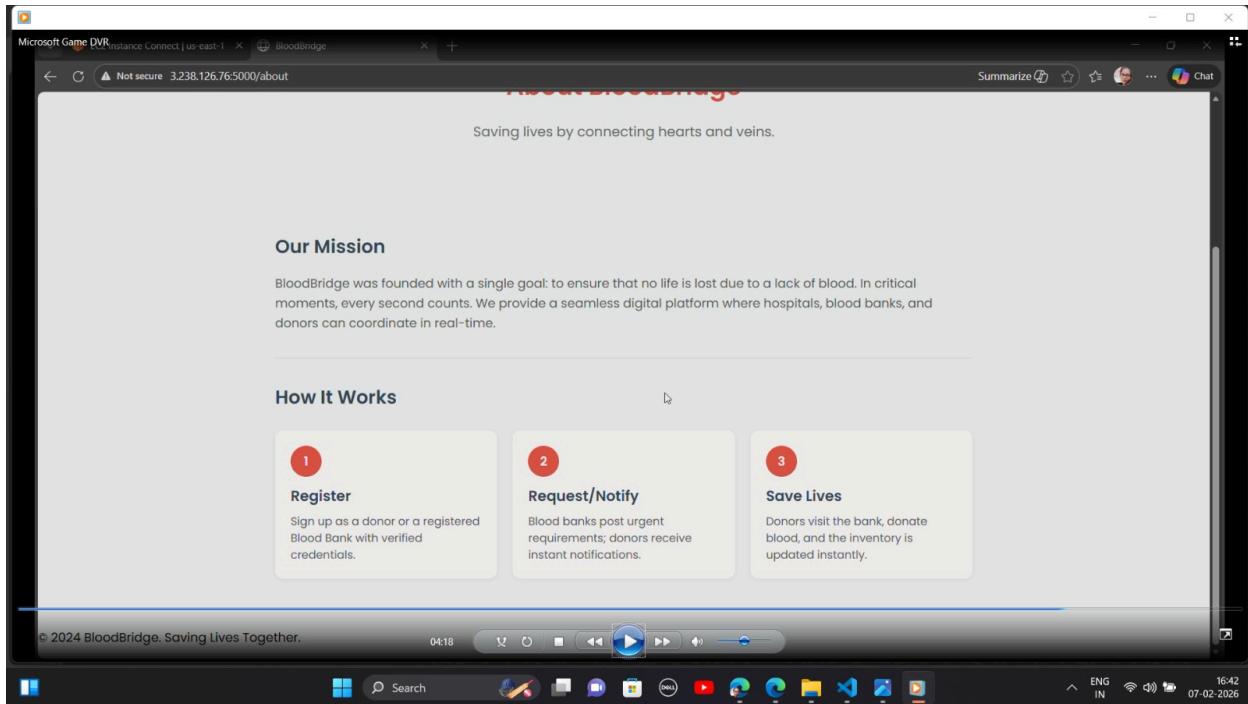
CloudShell Feedback © 2026, Amazon Web Services, Inc. or its affiliates. Privacy Terms Cookie preferences ENG IN 19:40 06-02-2026
```

Functional Testing To Verify The Project

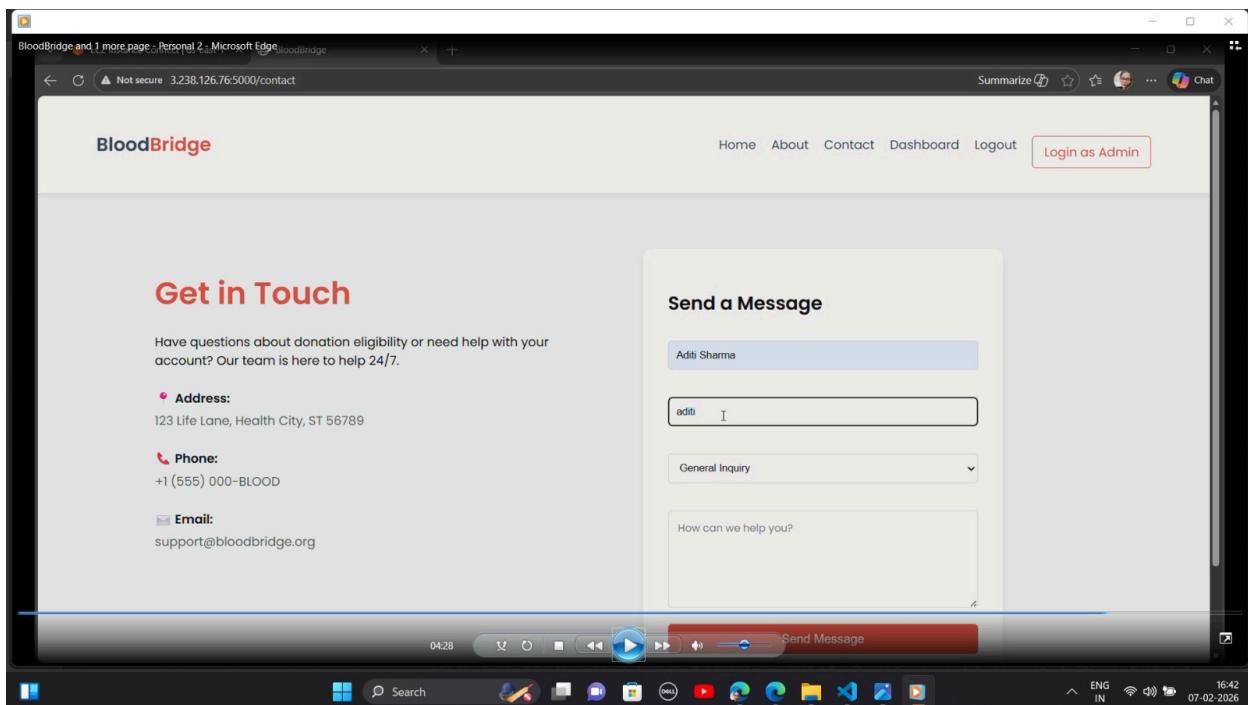
- Home Page:



- About Us page:



- **Contact Us page:**



- **Register page:**

Feb 6 21:10

BloodBridge

Home About Contact Login Sign Up [Login as Admin](#)

Join BloodBridge

Full Name

Email Address

Password

Blood Group

Admin Access Code (Optional)

Register

Already have an account? [Login here](#)

- **Recipient page:**

Microsoft Game DVR Instance Connect | us-east-1 | BloodBridge

Not secure 3.238.126.76:5000/dashboard

BloodBridge

Home About Contact Dashboard Logout [Login as Admin](#)

Welcome, Aditi (Recipient)

Request Blood

Patient Name

Blood Group Required

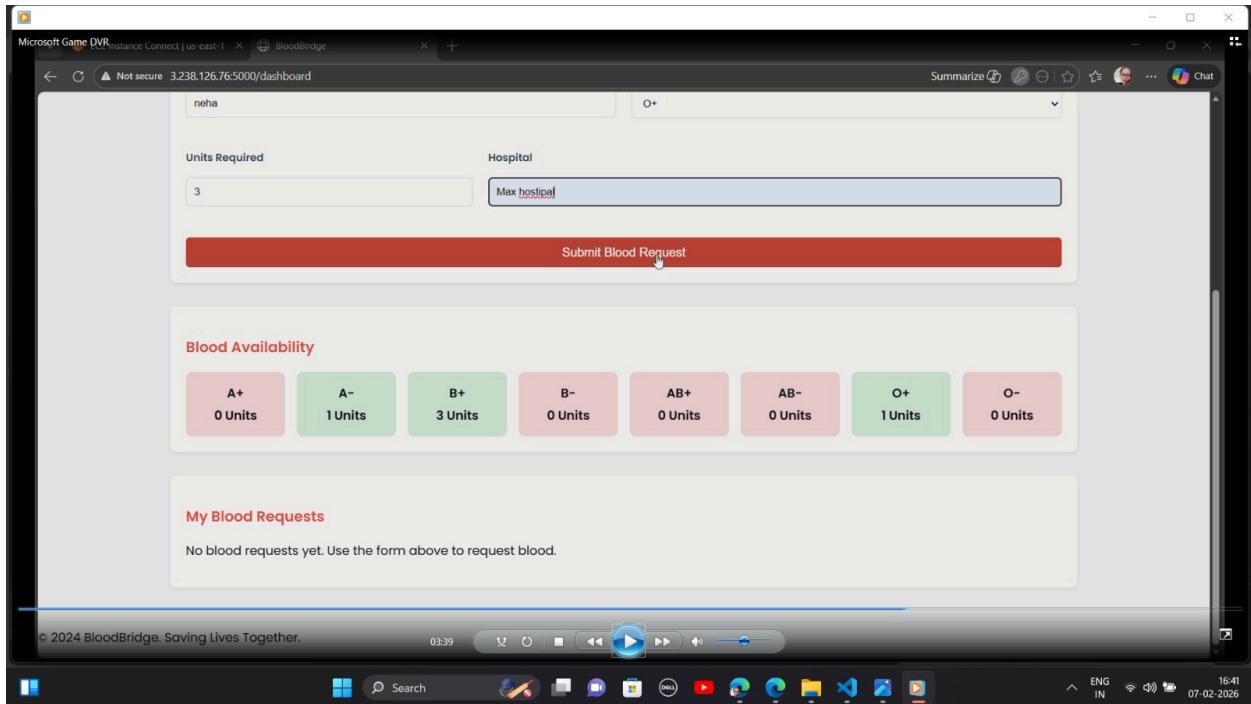
Units Required

Hospital

Submit Blood Request

Blood Availability 03:26

ENG IN 16:40 07-02-2026



Conclusion

This document presents a comprehensive guide for designing, developing, and deploying the **BloodBridge** platform using AWS services. It highlights the critical steps involved in configuring **Amazon DynamoDB**, building a **Flask-based web application**, and deploying the system on an **EC2 instance**. By following these milestones and implementation activities, the platform enables efficient management of blood donation requests while ensuring reliability and security.

Utilizing **EC2** for application hosting and **DynamoDB** for database management provides a scalable, high-performance cloud infrastructure capable of adapting to increasing user demand. The integration of AWS services not only enhances system availability and fault tolerance but also supports future expansion and optimization. Overall, BloodBridge demonstrates how cloud computing can be effectively leveraged to build a robust, real-world healthcare support application.

