CloudMed

Project submitted to the

APSSDC

Bachelor of Technology In

Computer Science and Engineering

Course: Cloud Computing DevOps

Submitted By

BATCH – 2ND

PARAMATA POOJITH RAJKAMAL - 24P35A0540

KANOORI RAMBABU - 23MH1A0597

SHAIK ISHAK BABA - 23MH1A05D0

MALLADI JOHN PRAKASH – 23MH1A05A9

ADITYA COLLEGE OF ENGINEERING AND TECHNOLOGY (ACET)

Under the guidance of

Mr.Anilkumar Varanasi,APSSDC

Mrs. Sumana Bethala, APSSDC

July-2025

ACKNOWLEDGEMENT

we would like to express our heartfelt gratitude to all those who have contributed to the successful completion of our summer internship project at Andhra Pradesh Skill Development Corporation (APSSDC). This opportunity has been an enriching and transformative experience for us, and I am truly thankful for the support, guidance, and encouragement we have received along the way. First and foremost, I extend my sincere regards to Mrs Sumana Bethala and Mr Anilkumar Varanasi, our supervisors and mentors, for providing us with valuable insights, constant guidance, and unwavering support throughout the duration of the internship. Their expertise and encouragement have been instrumental in shaping the direction of this project. we would like to thank the entire team at Andhra Pradesh Skill Development Corporation (APSSDC) for fostering a collaborative and innovative environment. The camaraderie, knowledge sharing, and feedback we received from the colleagues significantly contributed to the development and success of this project. In conclusion, we are honoured to have been a part of this internship program, and we look forward to leveraging the skills and knowledge gained to contribute positively to future endeavours.

Thank you.

Sincerely,

PARAMATA POOJITH RAJKAMAL - 24P35A0540

KANOORI RAMBABU - 23MH1A0597

SHAIK ISHAK BABA - 23MH1A05D0

MALLADI JOHN PRAKASH – 23MH1A05A9

1.ABSTRACT

This documentation outlines the design of a **scalable web platform** built on **Amazon Web Services (AWS)** to provide users with comprehensive medicine information, including uses, side effects, pricing, and provide certified links to buy medicines.

The platform leverages AWS cloud services to ensure **high availability, security, and real-time data access** while adhering to healthcare compliance standards.

The solution emphasizes **cost-efficiency, scalability, and regulatory**

compliance (e.g., HIPAA/GDPR) through AWS managed services. Future enhancements may incorporate AI-driven drug interaction warnings and supply-chain analytics.

This platform prioritizes **accessibility and trustworthiness**, ensuring that users—whether patients, doctors, or pharmacists—can quickly find accurate drug information, compare prices, and check availability without navigating multiple sources.

By leveraging AWS cloud infrastructure, the system ensures **high reliability, fast response times, and secure data handling**, making it a valuable tool for both everyday consumers and medical professionals.

This platform helps everyday users **easily find accurate medicine details**—like uses, side effects, and substitutes—in simple language. Anyone can quickly check where to buy medicines nearby at affordable prices. No technical knowledge is needed—just type the medicine name **(like 'Paracetamol' or 'Amoxicillin')**to get instant, reliable results.

2.INTRODUCTION

The healthcare industry faces significant challenges in providing **quick, reliable access to medicine information** and **authentic online purchase options**. Many patients struggle to find accurate drug details or trustworthy sources to buy medications, leading to confusion, delays in treatment, and even risks from counterfeit drugs.

To address these issues, this project introduces a **web-based Medicine Information & Procurement Platform**, deployed on **Amazon Web Services (AWS)**. The platform aims to:

- **Centralize verified medicine data** (uses, side effects, dosage) in one accessible location.
- **Provide direct purchase links** from authorized pharmacies to ensure safe transactions.
- **Offer a scalable, secure, and user-friendly solution** for patients, doctors, and caregivers.

By leveraging **AWS cloud infrastructure**, the system ensures high availability, fast performance, and seamless scalability—making critical medicine information and procurement options easily accessible to users worldwide.

2.1.Problem Statement:

In today's digital healthcare landscape, patients and medical professionals face three critical challenges:

1. **Fragmented Information**: Medicine details are scattered across multiple websites, often with inconsistent or outdated data
2. **Purchase Risks**: Many online pharmacy links lead to unverified sellers, raising concerns about medication authenticity
3. **Access Barriers**: Elderly users and rural populations struggle with complex interfaces and unreliable access

These issues create dangerous knowledge gaps, potentially leading to:

- Medication errors due to incorrect dosage information
- Financial losses from fraudulent online pharmacies
- Health risks from counterfeit or substandard drugs

Secure Cloud-Based Solution:

Our platform leverages AWS Virtual Private Cloud (VPC) architecture to deliver:

Core Security Framework:

- **Isolated network environment for all EC2 instances and RDS databases**
- **Multi-tier security groups controlling API and database access**
- **Encrypted data transit between components via VPC peering**

Performance Advantages:

- **Subnet-based traffic routing for low-latency global access**
- **NAT gateway integration for secure external pharmacy API connections**
- **Elastic Load Balancing within VPC for uninterrupted service**

2.2.Key Objectives:

1. **Military-Grade Data Protection**
    - VPC-bound resources with no public internet exposure for sensitive data
    - End-to-end encryption meeting HIPAA compliance standards
2. **Fault-Tolerant Access**
    - Multi-AZ deployment within VPC ensuring 24/7 availability
    - Automated backup systems with VPC flow logging
3. **Global Scalability**
    - Region-specific VPC configurations for localized medicine databases
    - Secure CDN integration through VPC endpoints

.

3.TECHNOLOGY STACK USED

3.1.EC2:

Role in Project: Primary compute service hosting the Node.js backend for medicine information processing and user authentication.

1. Core Configuration

- **Instance Type**: t2.micro (Free Tier eligible)
    - 1 vCPU, 1 GiB RAM
    - Burstable CPU performance
- **Operating System**: Amazon Linux 2023 AMI
- **Storage**: 8 GB EBS (SSD) root volume

2. Network Architecture

- **Virtual Private Cloud (VPC)**: Default lab VPC
- **Security Groups**:
    - HTTP/HTTPS access for web traffic
    - SSH restricted to lab IP range
- **Elastic IP**: Not used (Learner Lab limitation)

3. Integration with Other Services

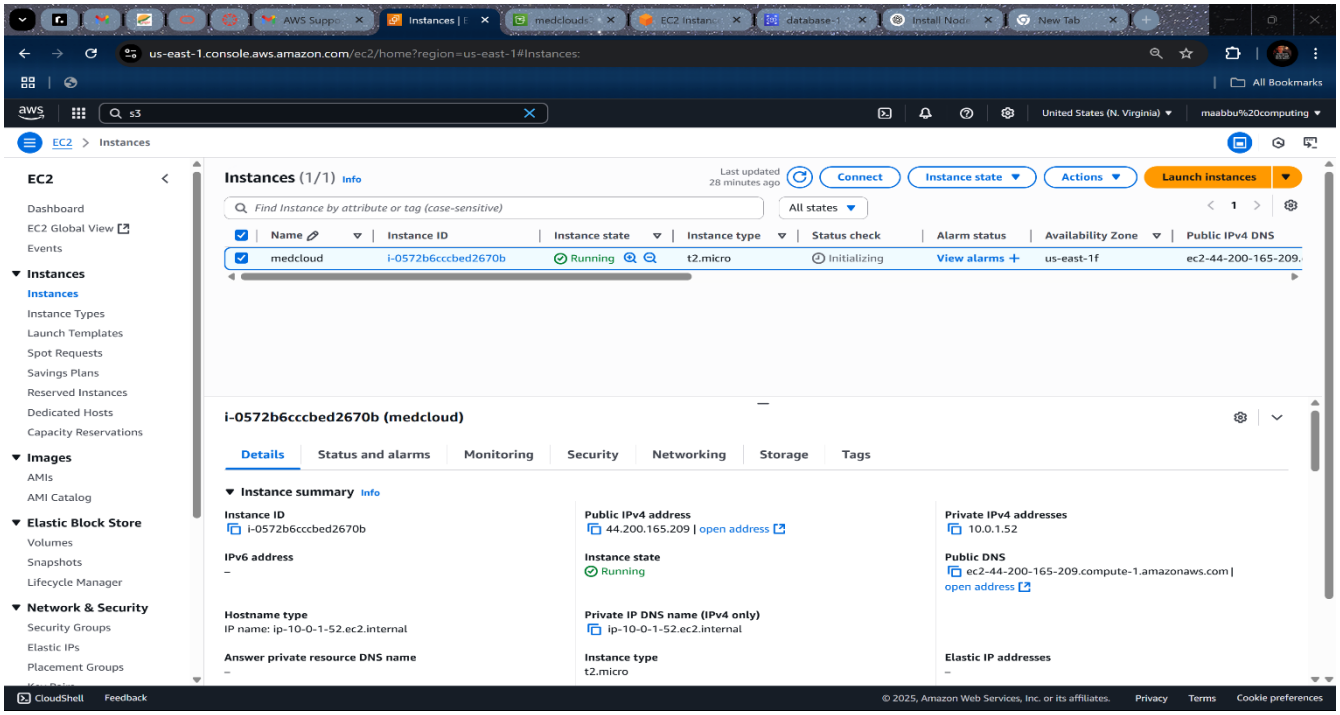
Service	Integration Purpose	Method
Amazon RDS	User authentication storage	MySQL connection pool
Amazon S3	Static frontend hosting	NGINX proxy_pass
CloudWatch	Basic performance monitoring	EC2 metrics dashboard

4. Adaptations

- **Persistence Strategy**:
    - Regular code backups to S3
    - PM2 process recovery on reboot
- **Cost Control**:
    - Strict Free Tier monitoring
    - Instance auto-stop during inactive periods

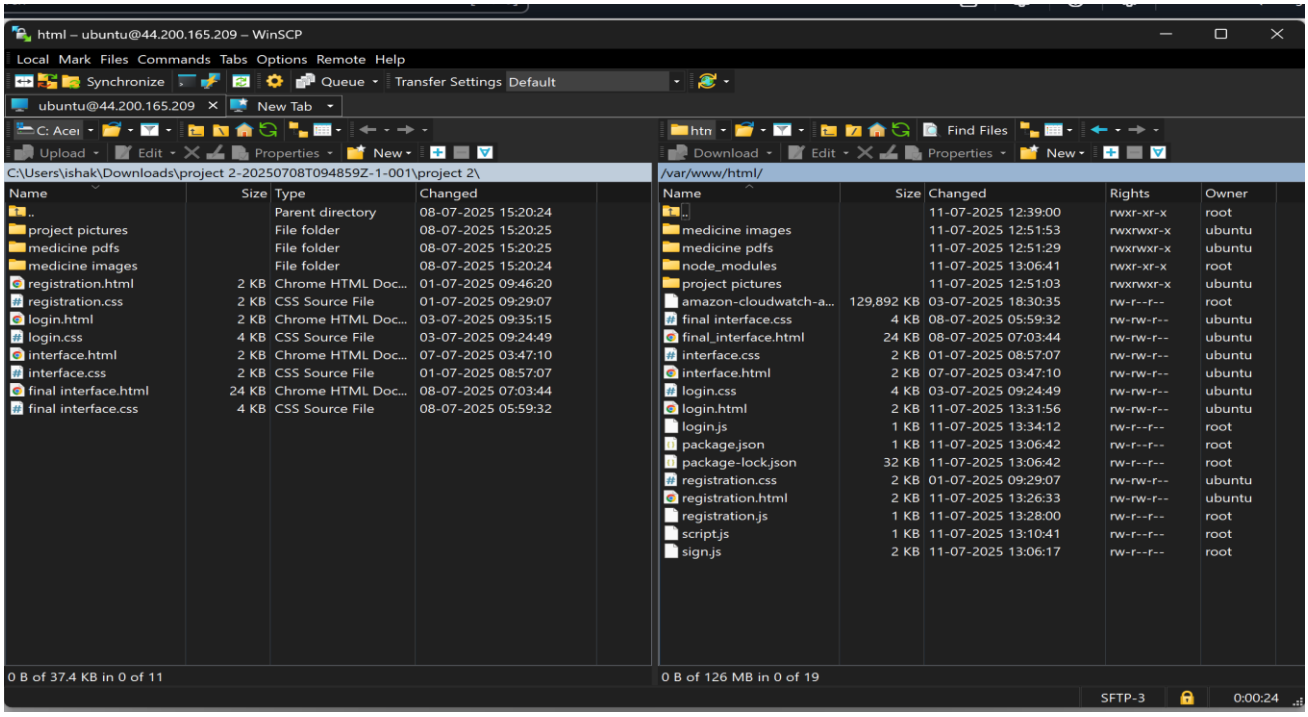
5. Security Implementation

- **IAM**: Root account IAM credentials
- **Data Protection**:
    - Environment variables for secrets
    - HTTPS enforced at apache level
    - HTTP
    - SSH



6.Usage of WINSCP

- This application is used for safe and secure transfer of program files and other assests like images , pdfs and other script files like server .



3.2.Amazon S3 Implementation:

Amazon Simple Storage Service (S3) provides secure, scalable object storage for static assets and backups in the medicine information platform. As a foundational AWS service, S3 ensures 99.999999999% data durability while maintaining cost-efficiency through Free Tier utilization.

Architectural Role

- **Primary Storage** for frontend static files (HTML/CSS/JS)
- **Central Repository** for medicine assets (drug images)
- **Backup Target** for EC2 application code and database snapshots

Critical Configurations

1.Bucket Properties

- Enabled static website hosting with index document support
- Configured regional storage in us-east-1 for lab consistency

- Implemented versioning for critical data recovery

2.Access Management

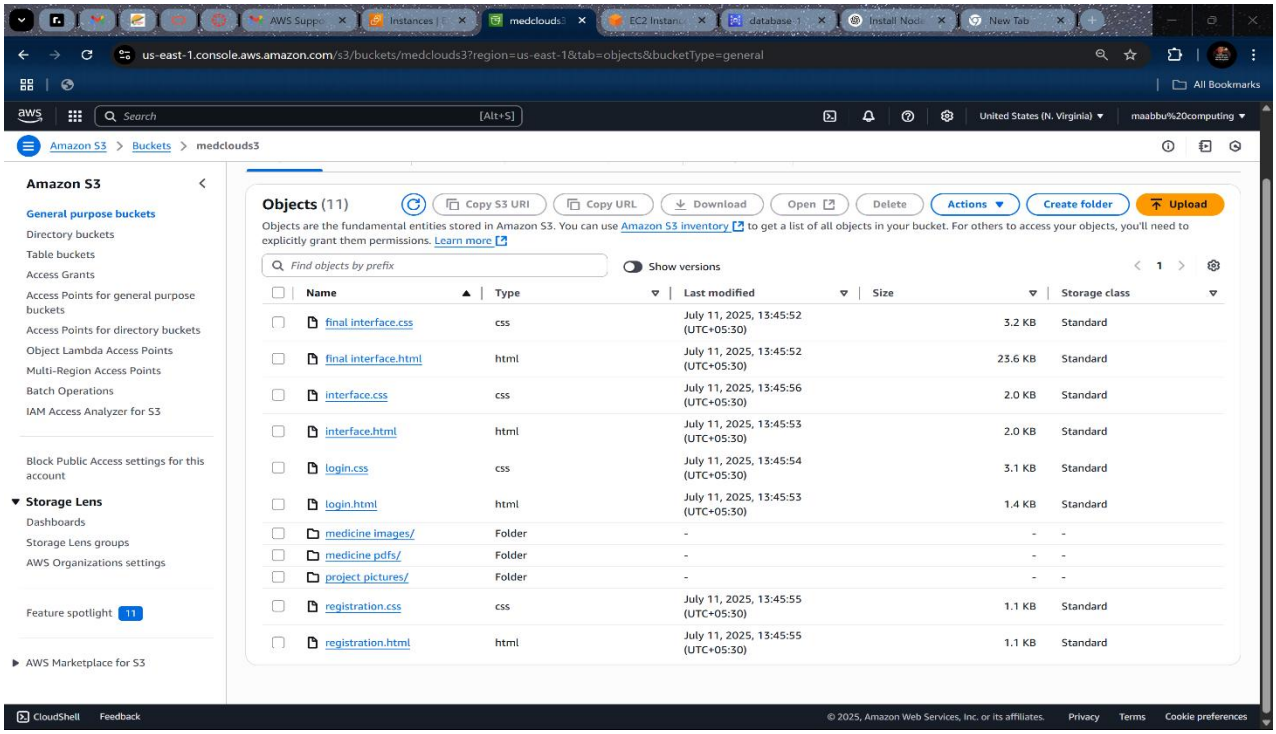
- IAM role-based permissions for EC2 write access
- Public read access restricted to /assets subdirectory
- Server-side encryption (SSE-S3) enabled by default

Security Implementation

- Bucket Policy: Explicit deny for unauthorized cross-account access
- CORS Configuration: Restricted to lab domain only
- Monitoring: S3 access logs archived in separate bucket

Future Enhancement Path

- Transition to S3 Intelligent-Tiering for cost optimization
- Implement presigned URLs for secure temporary access
- Enable Transfer Acceleration for global uploads



3.3 Amazon RDS Implementation:

Amazon Relational Database Service (RDS) serves as the secure, managed database backbone for storing all medicine information and user credentials. The MySQL-compatible deployment provides high availability within Learner Lab constraints while maintaining full ACID compliance.

Architectural Role:

- **Primary Data Store** for:
    - Medicine metadata (names, uses, side effects)
    - User authentication credentials
    - Pharmacy linkage information
- **Transaction Engine** for all CRUD operations
- **Reporting Source** for usage analytics

Deployment Configuration:

Instance Specifications

- **Engine**: MySQL 8.0 (Free Tier eligible)
- **Instance Class**: db.t2.micro (1 vCPU, 1GB RAM)
- **Storage**: 20GB General Purpose SSD

Adaptations:

- **Cost Control**:
    - Strict 20GB storage monitoring
    - Idle instance shutdown schedule
- **Connectivity**:
    - Public accessibility required (lab limitation)
    - IP restriction via security groups

Future Enhancement Path:

- Multi-AZ deployment for production readiness
- Read replica for reporting workloads
- Migration to Aurora for improved scalability

EC2 and RDS CONNECTIVITY :

This project demonstrates how to:

- Launch an **EC2 Ubuntu instance** to run a Node.js web server.
- Connect that server securely to an **Amazon RDS MySQL database**.
- Perform **basic CRUD operations** (like user registration and login) using **Express** and **MySQL** Node.js packages.

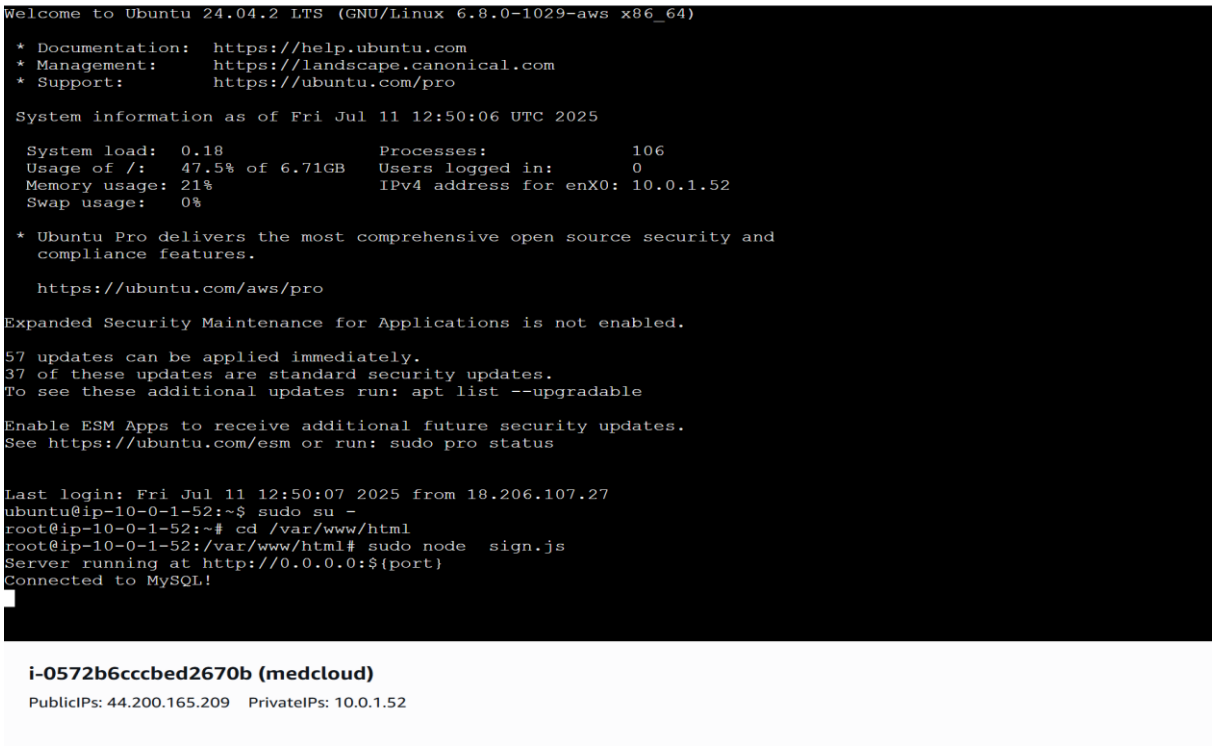
Components :

1. **Amazon EC2 Instance (Ubuntu)**
   Runs the Node.js server and serves static HTML/CSS/JS files.
2. **Amazon RDS (MySQL)**
   Manages the database (tables for users, etc.).
3. **Node.js Server (Express.js)**
   Handles HTTP requests (registration, login) and connects to RDS.

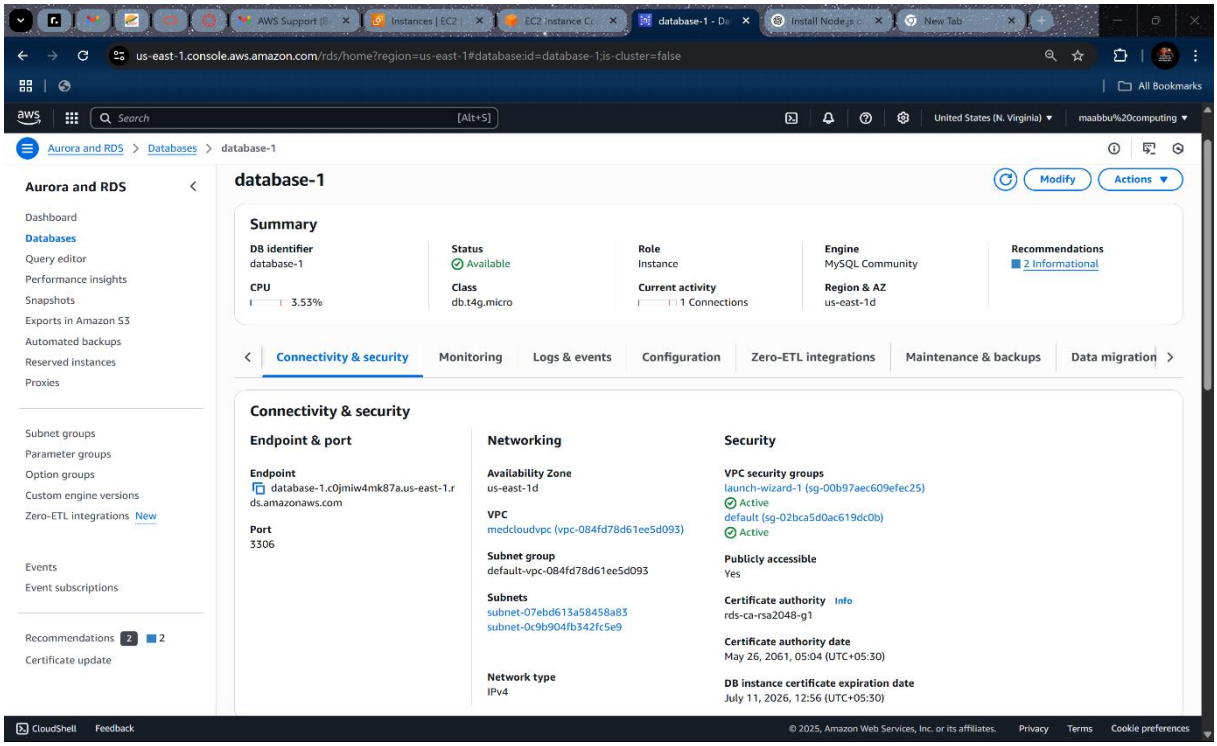
How the Connection Works

- **RDS** is created in the same **AWS region** as the EC2 instance.
- The **security group** for RDS allows inbound traffic **on port 3306 from the EC2 instance's IP** (or its security group).

- The Node.js server on EC2 uses the **MySQL Node.js driver** (mysql package) to connect to the RDS instance with **host, user, password, and database name**.



The server is connected to database



Database Dashboard after creation of database

Node.js code snippet model

- This server- side code is used for managing the RDS database , fetching user details ,posting them to database and , writing queries to SQL database .

```
const mysql = require('mysql');

const db = mysql.createConnection({
  host: 'your-rds-endpoint.amazonaws.com',  // ✅ RDS endpoint from AWS Console
  user: 'admin',                            // ✅ Your DB username
  password: 'YourPassword',                 // ✅ Your DB password
  database: 'your_database_name'            // ✅ Database you created
});

// Connect once on server start
db.connect(err => {
  if (err) {
    console.error('Database connection failed:', err);
    return;
  }
  console.log('✅ Connected to MySQL RDS successfully!');
});
```

The original code will accessible in implementation

(This is just a model to show, how RDS is connected using node.js server)

3.4.VPC (Virtual Private Cloud)

For this project, a custom Virtual Private Cloud (VPC) named **cloudmedvpc** was created to securely host the EC2 instance and the RDS database within an isolated network environment.

The VPC is divided into two subnets:

- **cloudmedvpc1** — used for public resources such as the EC2 instance running the Node.js server.
- **cloudmedvpc2** — used for private resources, specifically the Amazon RDS MySQL database.

To manage how traffic is routed inside the VPC, a custom route table named **medcloudroutetable** was configured. This route table includes a local route for internal communication within the VPC, as well as a default route (0.0.0.0/0) that directs outbound internet traffic through an internet gateway.

The internet gateway, named **medcloudnetgateway**, is attached to the VPC. This allows resources in the public subnet (cloudmedvpc1) to send and receive traffic from the internet. For example, when a user accesses the Node.js application hosted on the EC2 instance, the request enters through the internet gateway and is routed to the EC2 instance by the route table.

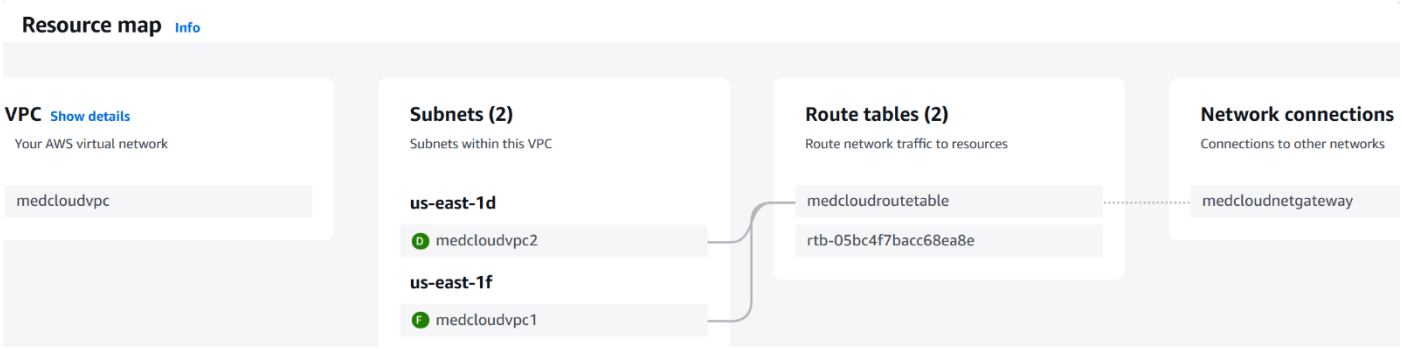
The EC2 instance's security group is configured to allow HTTP traffic on port 80 from any IP address, so the web application can be accessed publicly. In contrast, the RDS MySQL instance is deployed in cloudmedvpc2 and is secured to accept traffic only from the EC2 instance's security group on port 3306. This ensures the database is not exposed directly to the internet but remains reachable from the application server.

This VPC design ensures secure, controlled network communication between the web server and the database, with proper routing for public access where needed and private connectivity for backend operations. By separating public and private subnets, and using a dedicated route table and internet gateway, the network architecture balances accessibility with security, following AWS best practices.

In conclusion:

This setup ensures your project is:

- Secure and isolated
- Correctly routed for internet access



3.5. AWS CloudWatch

In this project, Amazon CloudWatch is used to monitor the health, performance, and logs of the EC2 instance named medcloud, which runs the Node.js web application connected to the RDS database.

AWS CloudWatch is a monitoring and observability service that collects metrics, logs, and events in real-time. By integrating CloudWatch with the medcloud EC2 instance, the project ensures that important performance data and system logs are automatically collected and can be analyzed to maintain availability and troubleshoot issues quickly.

To enable monitoring, the CloudWatch agent was installed and configured on the medcloud EC2 instance. This agent collects detailed operating system metrics such as CPU usage, memory utilization, disk usage, and network performance. Additionally, log files (such as application logs, web server logs, or system logs) are streamed to CloudWatch Logs, making it easy to view logs centrally in the AWS Console without needing to SSH into the server each time.

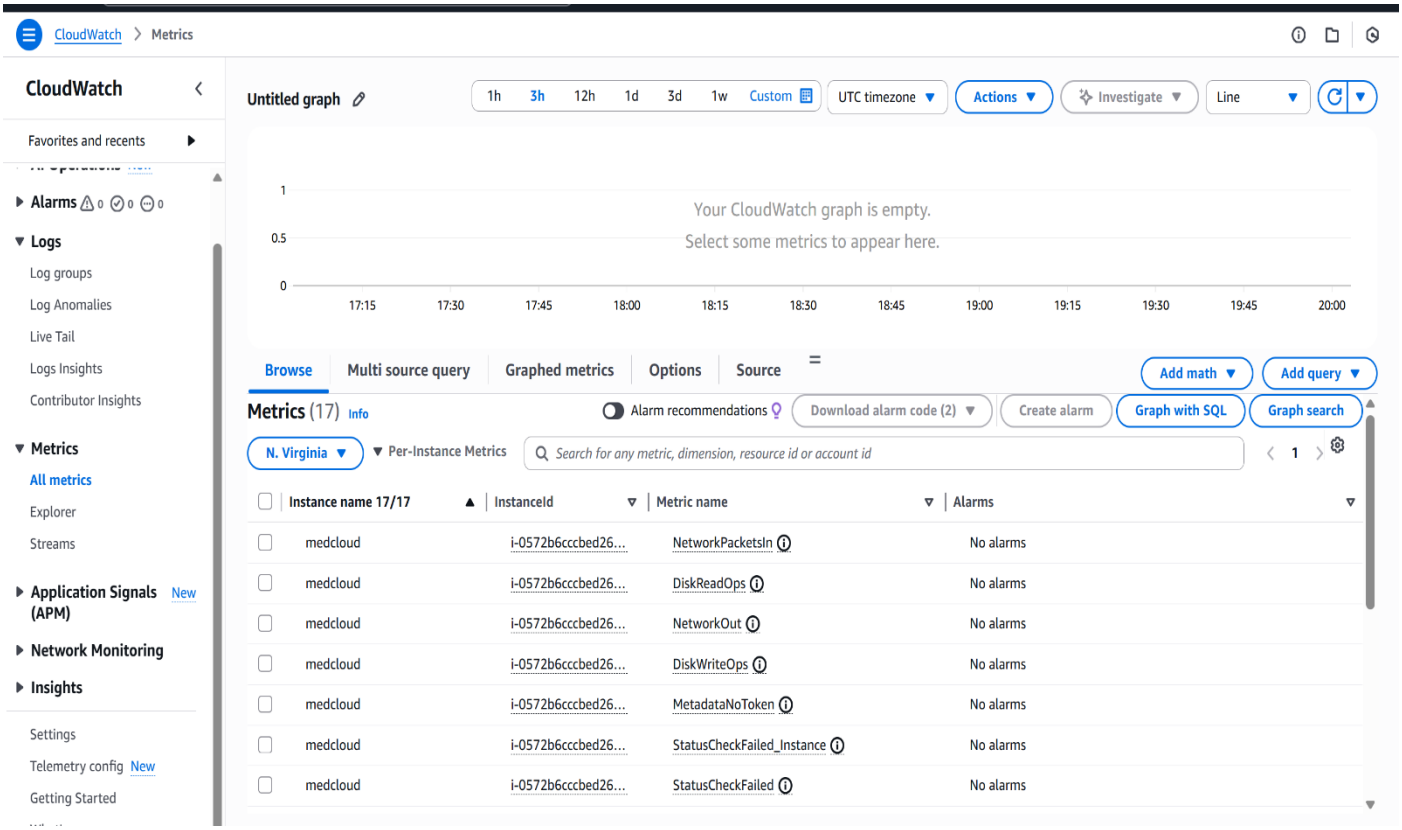
Using CloudWatch, alarms can be set up to notify the user if resource usage crosses critical thresholds. For example, an alarm can be configured to send an alert if the CPU utilization on the medcloud instance remains too high for a sustained period, or if available disk space drops below a safe level. This proactive monitoring helps maintain the application's availability and performance, and provides visibility into potential problems before they impact users.

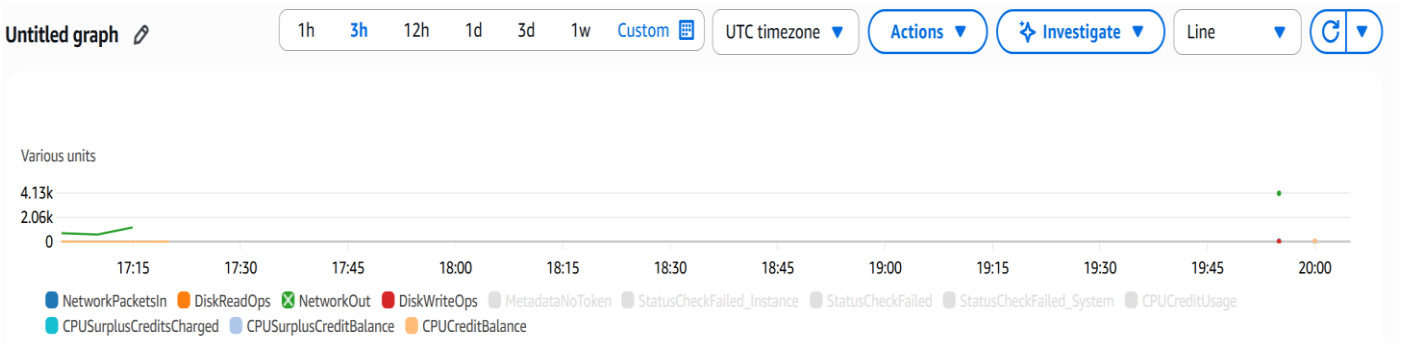
By combining CloudWatch metrics with detailed logging, the medcloud EC2 instance is fully integrated into AWS's best practices for operational monitoring.

This setup ensures the project is robust, observable, and easy to manage, supporting ongoing maintenance and future scaling needs.

Key points:

- The **CloudWatch agent** runs on the medcloud EC2 instance.
- It collects **system-level metrics** (CPU, RAM, disk, network).
- Important **log files** are pushed to **CloudWatch Logs**.
- This enables **real-time visibility** and **troubleshooting**.





4. System Design Architecture

This project is a web-based platform designed to provide everyday users and medical professionals with **easily accessible, trustworthy information about medicines**, along with **reliable purchase links to reputable online pharmacies**. The website is developed using **HTML, CSS, JavaScript, and Node.js**, and is deployed securely using **AWS cloud services**, following a robust VPC network architecture for high availability, security, and easy monitoring.

4.1.Key Technologies

- **Frontend:** HTML, CSS for layout and styling, JavaScript for interactivity.
- **Backend:** Node.js with Express.js handles HTTP requests, user registration, login, and serves web pages.
- **Database:** Amazon RDS (MySQL) stores user details and any related data.
- **Storage:** Amazon S3 may be used to store static resources like images or PDFs.
- **Monitoring:** AWS CloudWatch monitors EC2 performance, logs, and sends alerts if needed.
- **Network:** An isolated VPC with public subnets, route table, and internet gateway.

4.2.System Architecture

The architecture consists of the following components working together inside a custom **Virtual Private Cloud (VPC)** named **cloudmedvpc**.

1. **VPC & Subnets**
   - The **VPC** provides an isolated virtual network.
   - Two **public subnets** (cloudmedvpc1 and cloudmedvpc2) are created in different availability zones to support high availability.
   - These subnets host the EC2 instance and the RDS database, allowing the backend server to connect securely to the database within the VPC.

2. **Route Table & Internet Gateway**

- A **custom route table** (medcloudroutetable) handles network routing.
- An **internet gateway** (medcloudnetgateway) is attached to the VPC, providing public internet access to resources in the public subnets.
- This setup allows users to access the website from anywhere, while backend resources remain secured.

3. **EC2 Instance**

- The EC2 instance, named **medcloud**, runs the Node.js server.
- It serves the HTML, CSS, and JavaScript files to the user's browser.
- It handles incoming HTTP requests, processes form data, connects to the RDS database, and performs user operations such as registration and login.
- The EC2 instance is placed in the public subnet (cloudmedvpc1) and is secured using a security group that allows HTTP (port 80) and SSH (port 22, if needed) access.

4. **RDS MySQL Database**

- The RDS instance is deployed in the public subnet (cloudmedvpc2).
- The database securely stores user details and any other relevant data.
- The RDS security group allows inbound connections only from the EC2 instance's security group on port 3306 (MySQL), ensuring private communication.

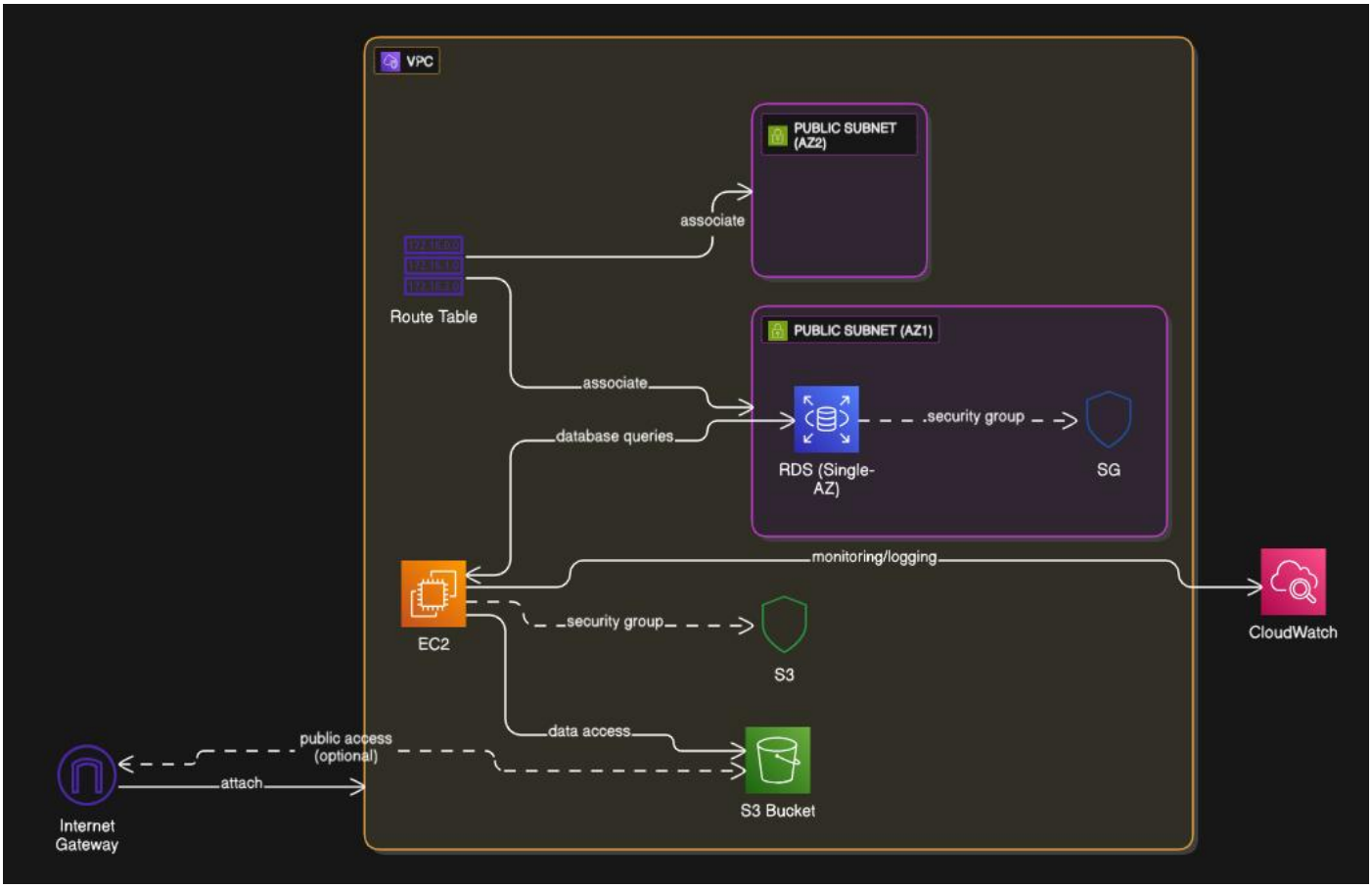
5. **S3 Bucket**

- An Amazon S3 bucket is available to store and serve static assets if needed (like booklets, medicine images, or PDFs).
- The EC2 instance can access this bucket to read or write static content.

6. **CloudWatch**

- **AWS CloudWatch** is integrated to monitor the medcloud EC2 instance.
- The CloudWatch agent runs on EC2, collecting metrics (CPU, memory, disk, network) and streaming logs (application logs, system logs) to the CloudWatch console.

- Alerts can be configured to notify administrators if performance thresholds are crossed.



System Architecture of CloudMed: EC2, RDS, S3 and CloudWatch in a Secure VPC

6. Implementation

6.1. Purpose:

The cloudmed platform aims to help normal users and medical professionals find reliable medicine information and trusted links to purchase from reputable online pharmacies.

Tech Stack:

- Frontend: HTML, CSS, JavaScript
- Backend: Node.js with Express.js
- Database: AWS RDS (MySQL)
- Networking: AWS VPC, Security Groups, Internet Gateway
- Monitoring: AWS CloudWatch

Key Implementation Features

6.2. User Registration.html

- Users fill a simple Sign Up form with First Name, Last Name, Email, Password.

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1" />
  <title>Registration Form</title>
  <link rel="stylesheet" href="registration.css" />
</head>
<body background="project pictures/sign up image.png">
  <div class="container">
    <h2>Create Account</h2>
    <form id="signupForm" action="/register" method="post">
      <div class="form-group">
        <label for="first-name">First Name</label>
        <input type="text" id="first-name" name="first_name" required />
      </div>
      <div class="form-group">
        <label for="last-name">Last Name</label>
        <input type="text" id="last-name" name="last_name" required />
      </div>
      <div class="form-group">
        <label for="email">Email Address</label>
        <input type="email" id="email" name="email" required />
      </div>
      <div class="form-group">
        <label for="password">Password</label>
        <input type="password" id="password" name="password" required />
      </div>
      <button type="submit" class="submit-btn">Register</button>
      <p id="responseMessage" style="margin-top:10px; color:white;"></p>
    </form>
  </div>
```

- Data is securely sent to the server.
- On success, the user is redirected to Login page.

Use short blocks only, then explain them in clear sentences.

- For example:

*“*The form uses an input with type="email" to validate user input on the client side.”

- Or:

“The Fetch API sends a POST request to /register with the form data encoded as URL parameters.”

Do not include your entire HTML or JavaScript files line-by-line in the main document.

- Instead, bundle the full files separately — for example:
    - Attach them in your project folder.
    - Or include them as an Appendix at the end of the PDF if your college or mentor wants the full code inside one file.

Client-Side Logic: registration.js

The JavaScript file:

- Listens for the form submit event.
- Prevents the default form submission to handle it via JavaScript.
- Gathers the form data using FormData and URLSearchParams.
- Uses the Fetch API to send a POST request to the /register endpoint on the server.
- Displays the server’s response in the responseMessage element and resets the form on success.

This design provides a smoother user experience by avoiding full page reloads and giving immediate feedback.

```
const form = document.getElementById('signupForm');
const responseMessage = document.getElementById('responseMessage');

form.addEventListener('submit', function (e) {
  e.preventDefault(); // Stop default form submission

  const formData = new FormData(form);
  const data = new URLSearchParams(formData);

  fetch('/register', {
    method: 'POST',
    body: data
  })
  .then(response => response.text())
  .then(result => {
    responseMessage.textContent = result;

    if (result.trim().toLowerCase().includes('successful')) {
      // If registration succeeded, wait 2 seconds and redirect to login page
      setTimeout(() => {
        window.location.href = 'login.html';
      }, 2000);
    } else {
      // If failed, do not redirect — keep message shown
    }

    form.reset();
  })
  .catch(error => {
    responseMessage.textContent = 'Error: ' + error;
  });
});
```

Explanation:

- Prevents the default form submission.
- Uses **Fetch API** to send the form data to the backend /register route.
- Displays server feedback.
- Redirects to the login page after successful registration.

Contains a form for new users to register:

- Fields: First Name, Last Name, Email, Password
- Submits via AJAX to /register endpoint

Server-side logic for Registration page:

```
app.post('/register', (req, res) => {
  const { first_name, last_name, email, password } = req.body;

  const sql = 'INSERT INTO users (first_name, last_name, email, password) VALUES (?, ?, ?, ?)';
  db.query(sql, [first_name, last_name, email, password], (err, result) => {
    if (err) {
      console.error('Error inserting:', err);
      return res.send('Error saving data');
 }
    console.log('User added:', result.insertId);
    res.send('Registration successful!');
  });
});
```

Explanation:

- Reads form data.
- Inserts a new user record into the users table.
- Sends confirmation to the client.

When the user submits the form:

- The Node.js server (with Express.js) handles the /register POST route.
- The server reads the incoming data, sanitizes it if needed, and inserts the new user record into the RDS MySQL database.
- The server responds with a success message or an error if something goes wrong (e.g., duplicate email).

6.4. Login.html

The login.html page provides a **secure, user-friendly interface** for registered users to log in to the MedInfo platform.
It acts as the **gateway** to the protected **medicine dashboard**, ensuring only valid users can access medicine information and trusted purchase links.

Key Elements

Form Fields

- **Username (Email):**
  The user enters the same email they used during registration.

- **Password:**
  The user provides their password.

✔ **Form Submission**

- The form is **handled with JavaScript (login.js)** using fetch().
- It **sends the credentials** to the backend route **/login** via **POST**.
- If the credentials are **valid**, the user is **redirected** to final_interface.html.
- If credentials are **invalid**, an error message appears **without reloading the page**

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
  <title>Simple Login</title>
  <link rel="stylesheet" href="login.css" />
</head>
<body background="project pictures/Screenshot (39).png">
  <div class="login-container">
    <h2>Login</h2>

    <form id="loginForm">
      <div class="form-group">
        <label for="username">Username</label>
        <input
          type="text"
          id="username"
          name="username"
          placeholder="Enter your username"
          required
        >
      </div>

      <div class="form-group">
        <label for="password">Password</label>
        <input
          type="password"
          id="password"
          name="password"
          placeholder="Enter your password"
          required
        >
```

Client-side login.js logic :

The login.js script handles the **client-side login logic** for the **MedInfo** platform. It ensures that user login is:

- Smooth and **dynamic**
- Error messages are shown **without reloading**
- Successful logins are redirected to the **protected dashboard** (final_interface.html)

```
const loginForm = document.getElementById('loginForm');
const loginMessage = document.getElementById('loginMessage');

loginForm.addEventListener('submit', function (e) {
  e.preventDefault();

  const formData = new FormData(loginForm);
  const data = new URLSearchParams(formData);

  fetch('/login', {
    method: 'POST',
    body: data
  })
  .then(response => response.text())
  .then(result => {
    if (result.trim().toLowerCase().includes('invalid')) {
      loginMessage.textContent = result;
    } else {
      // ✅On successful login, redirect to final_interface.html
      window.location.href = 'final_interface.html';
    }
  })
  .catch(error => {
    loginMessage.textContent = 'Error: ' + error;
  });
});
```

How it Works Workflow:

1. User clicks Log In.
2. login.js stops the default <form> submit.
3. Credentials are sent as a POST request via fetch.
4. Server checks the credentials in AWS RDS MySQL.
5. Server replies with success or invalid login.
6. login.js handles this reply to:

- Show error message or
- Redirect to final_interface.html

Features

- Uses Fetch API for modern AJAX calls.
- No page reloads — fast user experience.
- Provides instant feedback for wrong credentials.
- Keeps the app clean and user-friendly.

Server-side login logic:

```
app.post('/login', (req, res) => {
  const { username, password } = req.body;

  console.log('Incoming:', username, password);

  const sql = 'SELECT * FROM users WHERE email = ? AND password = ?';
  db.query(sql, [username.trim(), password.trim()], (err, results) => {
    if (err) {
      console.error('Error checking login:', err);
      return res.send('Error during login');
    }

    console.log('Query results:', results);

    if (results.length > 0) {
      res.redirect('/final interface.html');
    } else {
      res.send('Invalid username or password');
    }
  });
});
app.listen(port, '0.0.0.0', () => {
  console.log('Server running at http://0.0.0.0:${port}');
});
```

Explanation:

- Checks credentials against the database.
- On success, redirects the user to final interface.html.
- Otherwise, sends an error message.

Server Configuration

Server Runs On:

- **Node.js** with **Express**
- Port 80 (0.0.0.0 for public access)
- Connected to AWS **RDS** (MySQL)

How It Works

Receives Login Request

- Triggered when a user submits the **login form**.
- The **POST request** is sent to **/login** by the login.js script via fetch().

Extracts Credentials

- Reads username (email) and password from req.body.

Runs SQL Query

Uses mysql to query the users table:

SELECT * FROM users WHERE email = ? AND password = ?

Validates User

- If a match is found:
  - Sends a success response (Login successful!).
- If no match:
  - Sends Invalid username or password.

Frontend Handles Response

- If valid → login.js redirects user to final_interface.html.
- If invalid → shows error under form.

6.3.Final_Interface.html Example block:

```
</div>
<div class="inner-div">
    <div><img src="medicine images/Glipizide.jpeg" width="150px" height="250px" style="border-radius: 10px;"></div>
    <div style="font-family: Roboto, Arial; color: black; font-size: 18px; padding:3px 0px 3px 0px;">
        Medicine : Glipizide
    </div>
    <div><a href="medicine pdfs/Glipizide - Blood Sugar Control Medicine.pdf" target="_blank"><button class="infoButton">More Info&#8594</button></a></div>
    <div style="margin-bottom: 10px;">
        <a href="https://www.netmeds.com/catalogsearch/result/glipizide/all">
            <h4 style="color: black; font-style: Roboto, Arial;">
                Click Here To Purchase
            </h4>
        </a>
    </div>
</div>
```

final_interface.html

- Organized by **categories** like *Diabetes* and *Hypertension*
- Each medicine has:
  - Name & Image (medicine images/)
  - Link to PDF (medicine pdfs/) with reliable description.
  - Button to purchase via **Netmeds** or other reputable sources.
  - Categories are well defined
  - Images and PDFs organized
  - Buy links open in new tab

Summary of Implementation:

Fully working flow:

Register ➜ Save to RDS ➜ Login ➜ Verify ➜ Serve main interface ➜ Show medicine info & purchase

Stack:

HTML + CSS + JavaScript (Client) | Node.js (Server) | MySQL RDS (DB) | AWS (Infra)

7.Results/Outputs

7.1. Registration Page

Description:

This screenshot demonstrates the **registration form** on the MedInfo website. It shows a user named *Ishak Baba* filling out the **Create Account** form. The background image illustrates a pharmacy shelf, visually reinforcing the theme of a medical information site.

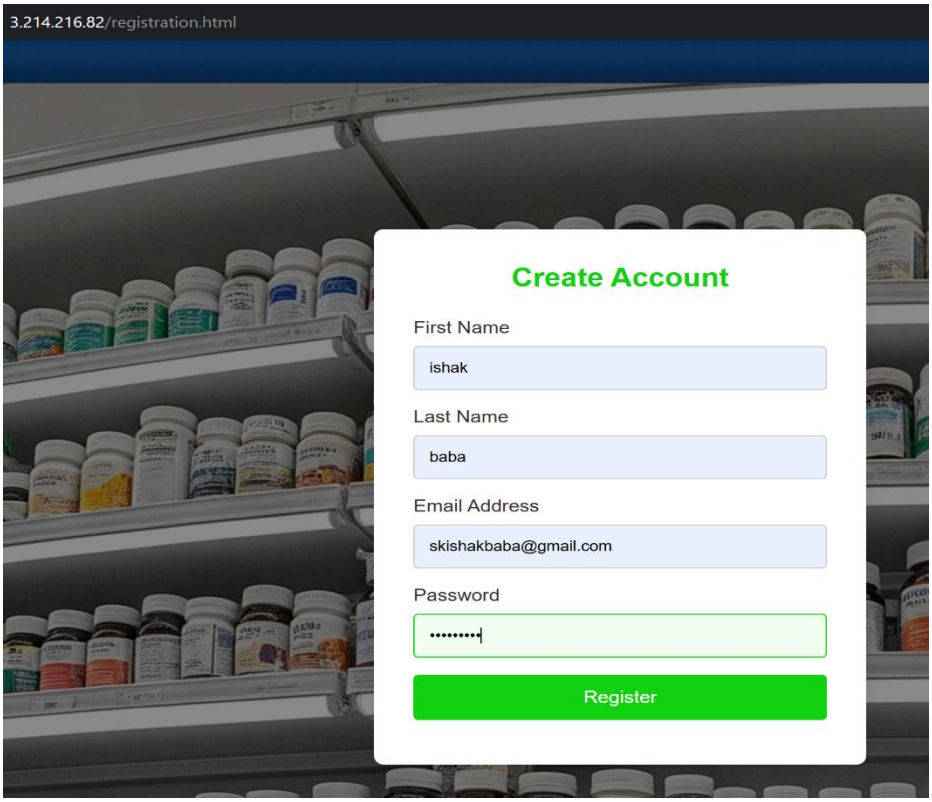
Fields Filled:

- **First Name:** ishak
- **Last Name:** baba
- **Email Address:** skishakbaba@gmail.com
- **Password:** (Hidden for security)

Expected Result:

When the user clicks **Register**:

- The **registration form** data is **sent to the server** (/register endpoint).
- The Node.js backend **inserts** this user's details into the **MySQL database**.
- The form displays a **confirmation message** ("Registration successful!") if the operation succeeds.
- The user is then **redirected** to the **login page** automatically after a short delay (handled in registration.js).

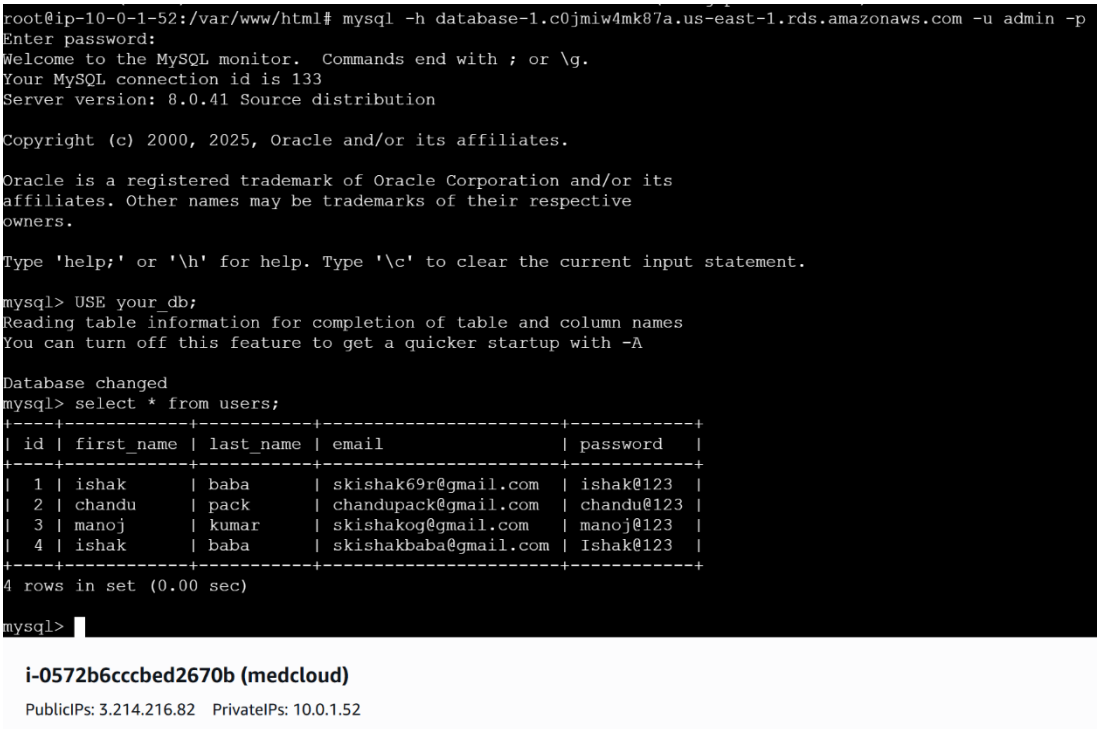


Purpose:

This confirms that:

- The **front-end form** properly **collects user input**.
- The **connection to the backend** and database **works correctly**.
- The **visuals** and **validation** of the page align with the project's goal: an accessible, user-friendly platform for new users to **sign up** securely.

Database receives the user details:



The screenshot shows that after a user completes the registration form, their details are successfully sent to the MySQL database running on the AWS RDS instance.

Explanation:

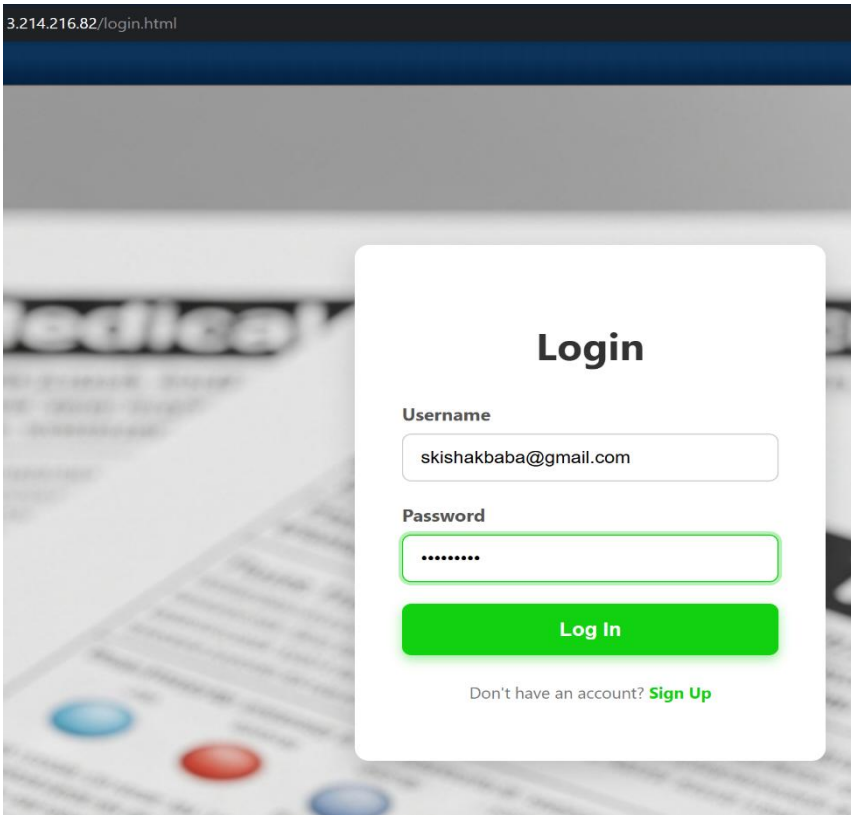
- When the **user submits the registration form**, the registration.js script sends the form data via a **POST request** to the /register route handled by the **Node.js server**.
- The server's /register endpoint runs an **INSERT SQL query** to store the new user's:
  - first_name
  - last_name
  - email
  - password

- The screenshot confirms that the new data is added to the **users table** and can be retrieved with a SELECT query.
- This verifies that the **EC2 instance**, **Node.js server**, and **RDS database** are working together as expected.

Observation:

- Each row in the table corresponds to a **registered user**.
- The data matches what was entered in the **registration for**

7.2.Login Page



Description:

This screenshot shows the Login form of the MedInfo website. It provides a simple, clear interface for registered users to sign in securely. The clean form appears on top of a medical-themed background, maintaining brand consistency.

Fields Available:

- **Username:** The user enters their registered email address (which acts as the username).
- **Password:** The user enters their account password.

User Action:

- The user fills in valid credentials and clicks the bright green Log In button.
- If the credentials match an entry in the MySQL database, the Node.js backend validates the login.
- On success, the JavaScript (login.js) handles redirecting the user to the final_interface.html page, where they can browse medical information and buy links.

Expected Results:

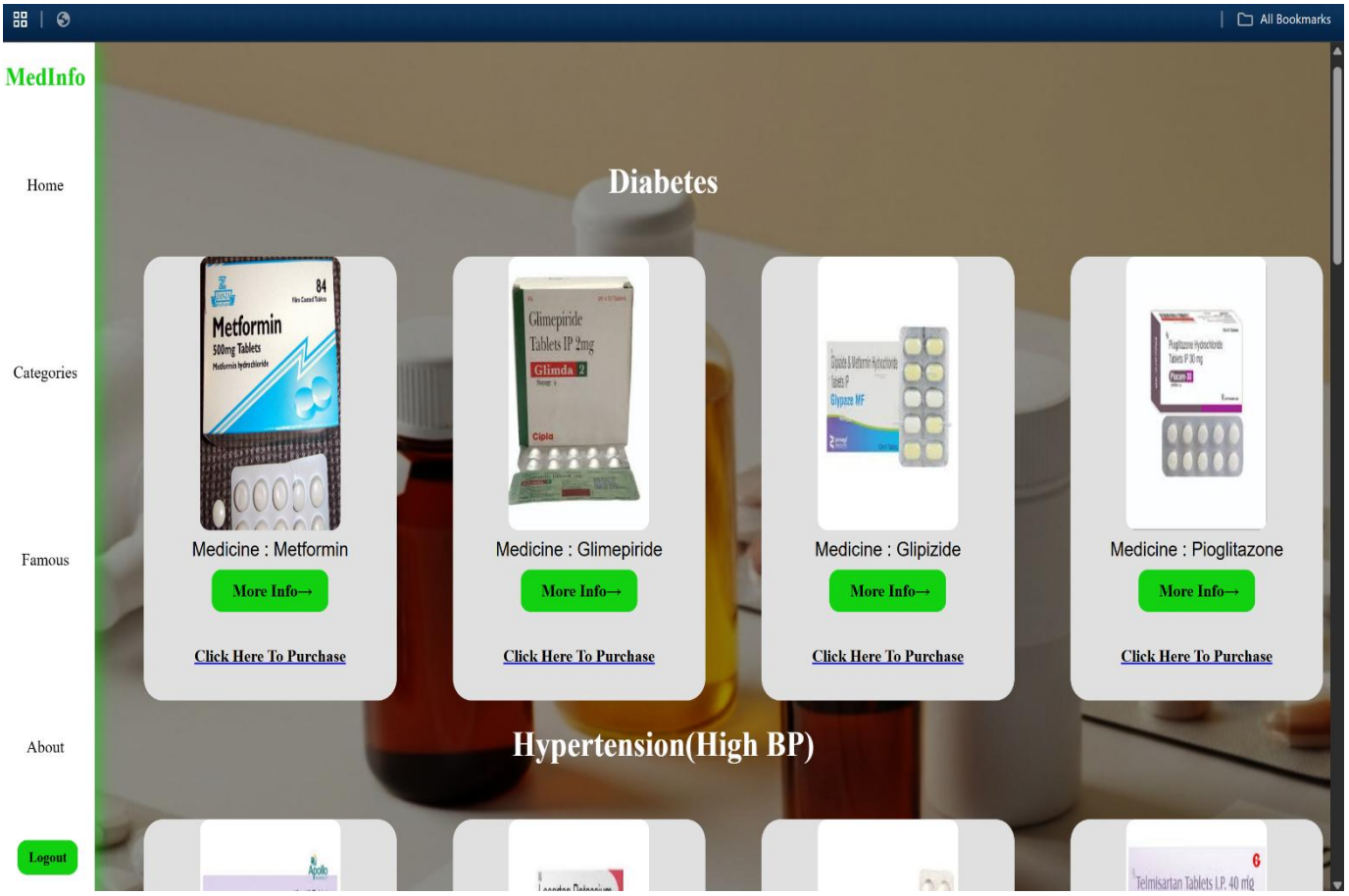
- **Successful Login:**
  - Valid credentials → Server verifies → User is redirected to the main interface.
  - No error messages appear.
- **Invalid Login:**
  - Wrong username/password → Error message appears below the form, indicating Invalid username or password.
  - User stays on the same page to try again.

To show what happens after login :

Server-Side Credential Check:
The server (Node.js + MySQL) runs a query to check if the credentials exist in the users table of the database.

- **If the credentials match:**
  - The server responds with Login successful!
  - The client-side script detects this and redirects the user to final_interface.html — the main page with medicine information and purchase links.
- **If the credentials do not match:**
  - The server responds with Invalid username or password.
  - The client shows this message without redirecting, so the user can try again.

7.3.final_interface Page



Final User Interface

Purpose:
This screenshot demonstrates the successful final output of the MedInfo web application once a user has logged in using valid credentials.

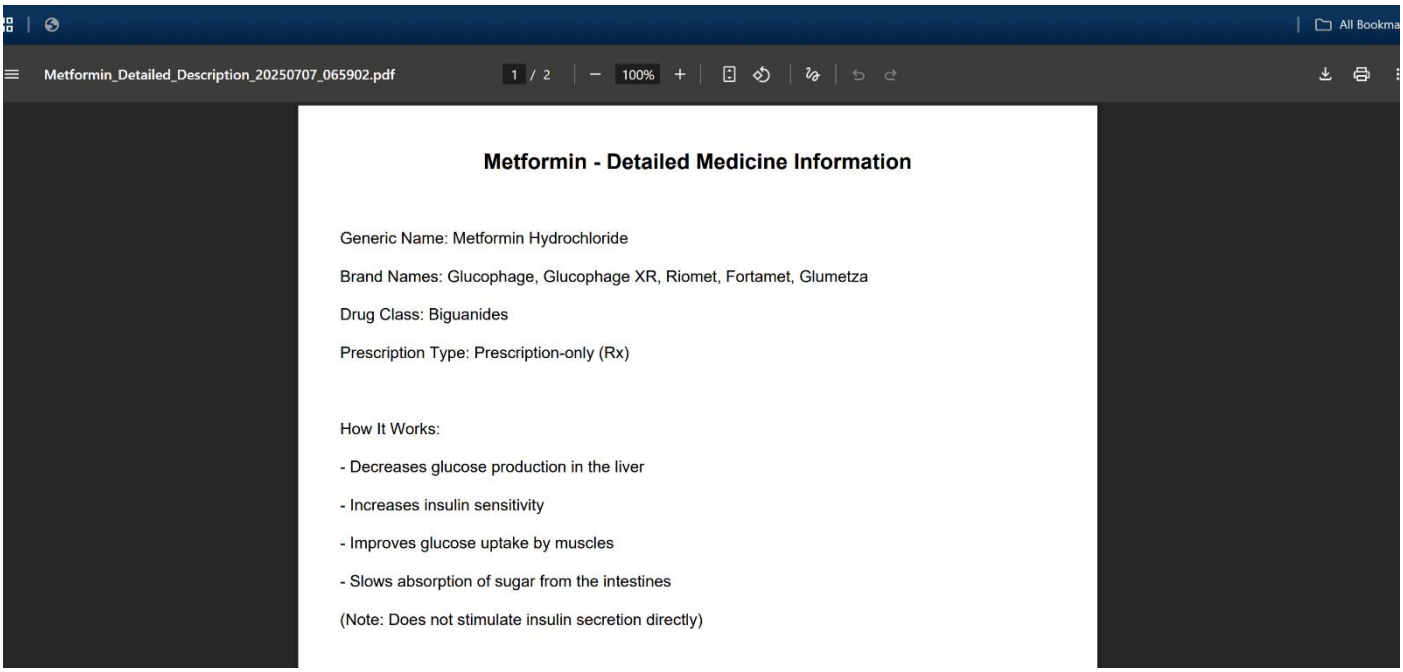
Description:

- After logging in, the user is redirected to the main dashboard page (final_interface.html).
- The page displays curated medicine information, neatly categorized (example shown: Diabetes and Hypertension (High BP)).
- Each medicine block shows:
    - Medicine Image
    - Name
    - 'More Info' button that opens a detailed PDF.
    - Trusted purchase link (e.g., Netmeds, Apollo) for direct buying.
- The sidebar navigation provides quick access to:
    - Home
    - Categories
    - Famous
    - About

Logout button (to return to login or exit securely)

Key Outcomes:

- Verifies that user authentication works: only valid users reach this page.
- Shows that assets (images, PDFs) are loaded correctly.
- Demonstrates the project goal — providing easy access to reliabl
- medicine information and trusted purchase links.



Detailed Medicine Information PDF + Trusted Purchase Links

Purpose:

This output screenshot shows the medicine details PDF that is generated when the user clicks “More Info →” on the final interface page. It demonstrates that the website does more than just list medicine names — it delivers clear, verified medical information that users can read, download, or print.

Description:

- The sample file (Metformin_Detailed_Description) contains:
  - Generic Name
  - Common Brand Names
  - Drug Class
  - Prescription Type
  - An easy-to-understand “How It Works” section.
- This format ensures clarity for general users and reliability for professionals.
- The information is presented in a clean PDF, making it suitable for saving or sharing.

Trusted Purchase Links:
Next to every More Info button, the interface also provides a direct, reputable link (e.g., NetMeds, Apollo Pharmacy, or other trusted e-commerce sites).

- These links are clearly marked as “Click Here To Purchase”.
- They ensure that users do not get misled by unsafe or unverified sources.
- This bridges the gap between medical awareness and safe access to real products.

Key Benefits:
Users understand what the medicine is, how it works, and if it fits their need — before buying.

The links connect to reputable pharmacies, so users feel safe making purchases online**.**
This output proves that your project combines information + practical access, which is exactly what your goal states: “helping normal users and professionals get medicine info and find reliable purchase options.”

Result Purpose:
The PDF and purchase link together demonstrate that your system is more than static pages — it is a practical, functional health resource, powered by HTML, CSS, JS, Node.js, connected to AWS RDS for user data, and secure online sources for buying.

8.Conclusion & Future Scope

8.1.Conclusion

This project successfully demonstrates how to build a basic yet practical medicine information portal using HTML, CSS, JavaScript, Node.js, and a MySQL database hosted on AWS RDS.

- The registration and login system ensures only authorized users access the medicine details.
- The final interface page presents categorized medicines with clear visuals, downloadable PDFs, and direct purchase links to trusted medical websites.
- The backend securely handles user data and connects to the database using EC2 and VPC components, following good cloud architecture practices.
- With CloudWatch monitoring, the system is also prepared for basic logging and usage insights.

This proves how cloud resources can be combined with simple web technologies to deliver real-world health information in a safe and user-friendly manner.

8.2.Future Scope

- To expand and improve this project, the following enhancements can be considered:
- Secure Authentication: Add hashed passwords and OTP/email verification
- to protect user data even better.
- Admin Dashboard: Build an admin panel to add, edit, or remove medicine entries, so the database stays up to date.
- Search & Filter: Add a live search bar and filters for disease categories, dosage forms, brands, etc.
- Advanced Monitoring: Use more AWS services like CloudTrail, AWS Config, or GuardDuty for security audits and compliance.

- User Feedback: Allow users to leave ratings or reviews for medicines or share usage experiences to help others.
- Data Analytics: Track which medicines are most viewed or downloaded, giving insights for further improvements**.**
- Multi-Language Support: Add translations so users can read the same information in their preferred language.
- Integration with Pharmacies: Directly connect with verified pharmacies for live stock availability or order tracking.

- Overall, this project lays a strong foundation for a safe, informative, and accessible **medicine information portal**, demonstrating practical skills in **full-stack development**, **cloud deployment**, and **secure database connections** — ready to be expanded into a more advanced real-world solution.

REFERENCES:

- Analytics Vidhya. (2022, June). Data Analysis Project for Beginners using Python. Retrieved from https://www.analyticsvidhya.com/blog/2022/06/data-analysis-project-for-beginners-using-python/

- Make Me Analyst. (n.d.). Python Libraries for Data Analysis. Retrieved from https://makemeanalyst.com/data-science-withpython/python-libraries-for-data-analysis/

- Reback, J., McKinney, W., Jbrockmendel, & L, Perktold, J., Seabold, S., Stéfan van der Walt. (2020). pandasdev/pandas: Pandas v1.2.4. Zenodo. Retrieved from https://doi.org/10.5281/zenodo.3632209