

Fully Automated Scalable CICD Pipeline for Serverless Django on AWS

Table of Contents

1.	Introduction.....	2
2.	Pre-requisites.....	2
3.	Summary.....	2
4.	Procedure Module wise	
1.	Setting up a VPC and an EC2 instance for a Jenkins server with the necessary configuration.....	3
2.	Set up an RDS database, configure it appropriately, and create the necessary database and tables.....	8
3.	Setting up a secure web application using Route 53, a Load Balancer, and ACM.....	12
4.	Set up ECR, ECS, task definition, and its service for containerized deployment.....	21
5.	Overview of the Web Application and Implementing Necessary Modifications.....	29
6.	Configure the Jenkins console and set up its pipeline code.....	33
7.	Test and Validating the Setup.....	39
5.	Overall, of this project.....	45

1. Introduction:

This document outlines the steps to configure the serverless automation architecture.

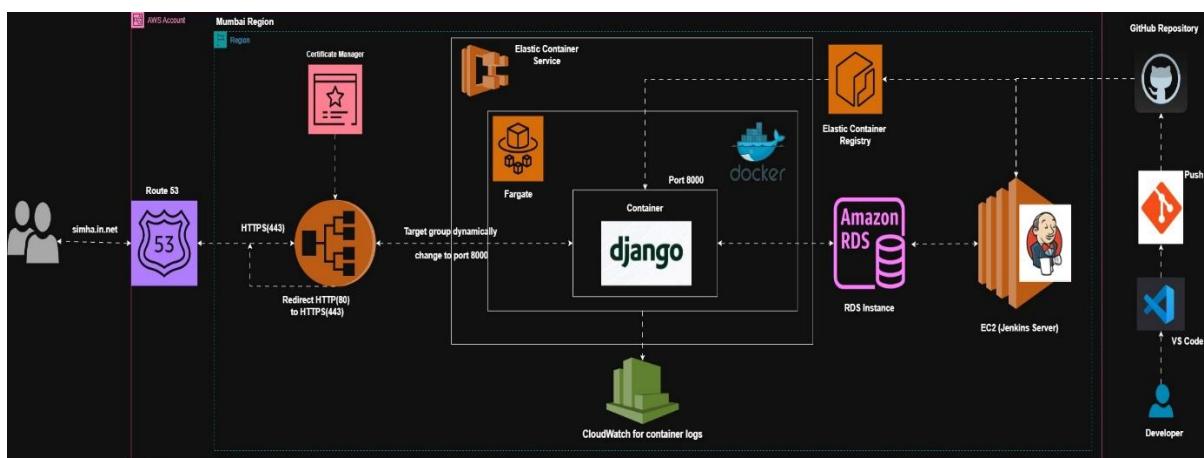
2. Pre-requisites:

Appropriate Access/Credentials to login to the AWS Console.

- AmazonEC2FullAccess
- AmazonVPCFullAccess
- AmazonRDSFullAccess
- AmazonECSFullAccess
- AmazonElasticContainerRegistryPublicFullAccess
- AWSCertificateManagerFullAccess
- CloudWatchEventsFullAccess
- AmazonRoute53FullAccess

3. Summary:

This architecture represents a fully automated and scalable CI/CD pipeline for a Django application on AWS using rolling updates for seamless deployments. The process begins with a developer pushing code from VS Code to GitHub, which triggers Jenkins on an EC2 instance to automate the build and deployment workflow. Jenkins pulls the latest code, builds a Docker image, and pushes it to Amazon Elastic Container Registry (ECR). AWS ECS (Fargate) then pulls the updated image and deploys it within containers running Django on port 8000, dynamically managed by an Application Load Balancer (ALB), which routes HTTP (80) traffic securely to HTTPS (443). The application interacts with an Amazon RDS database, with logs centrally monitored via CloudWatch. Route 53 manages domain resolution, ensuring seamless traffic distribution. Rolling updates are used with a minimum of 100% running tasks and a maximum of 200% to guarantee zero downtime. This setup enables automated deployments, high availability, and scalability, ensuring a secure, efficient, and fully optimized DevOps pipeline with minimal manual intervention. 



4. Procedure Module wise:

Module 1

Setting up a VPC and an EC2 instance for a Jenkins server with the necessary configuration

1. Create a VPC with a CIDR block of “172.31.0.0/16” in the Mumbai region.

The screenshot shows the AWS VPC console interface. At the top, there's a search bar and a table titled "Your VPCs (1/1)". The table has columns for Name, VPC ID, State, Block Public Access, and IPv4 CIDR. One row is selected, showing "ApplicationVPC" as the name, "vpc-0ab9222cd31d02273" as the VPC ID, "Available" as the state, "Off" as Block Public Access, and "172.31.0.0/16" as the IPv4 CIDR. Below this, a detailed view for "vpc-0ab9222cd31d02273 / ApplicationVPC" is shown. It includes tabs for Details, Resource map, CIDRs, Flow logs, Tags, and Integrations. The Details section contains various configuration parameters like VPC ID, State, Block Public Access, DNS hostnames, and Main route table.

Name	VPC ID	State	Block Public...	IPv4 CIDR
ApplicationVPC	vpc-0ab9222cd31d02273	Available	Off	172.31.0.0/16

Details

VPC ID vpc-0ab9222cd31d02273	State Available	Block Public Access Off	DNS hostnames Enabled
DNS resolution Enabled	Tenancy default	DHCP option set dopt-003a667ff807d0eaf	Main route table rtb-0de2905abfc36a848
Main network ACL acl-0eb36cef045750a4e	Default VPC Yes	IPv4 CIDR 172.31.0.0/16	IPv6 pool -

2. Create three subnets within the VPC, each corresponding to a different availability zone, and modify their settings to allow the assignment of public IP addresses.

Subnets (3) Info

Last updated 1 minute ago [Actions](#) [Create subnet](#)

Name	Subnet ID	State	VPC	Block Public...	IPv4 CIDR	IPv6 CIDR
ApplicationSubnet1	subnet-0c77452e16c19ded6	Available	vpc-0ab9222cd31d02273 App...	<input type="radio"/> Off	172.31.0.0/20	-
ApplicationSubnet2	subnet-053819e25328c54e9	Available	vpc-0ab9222cd31d02273 App...	<input type="radio"/> Off	172.31.32.0/20	-
ApplicationSubnet3	subnet-05efffab42df7892a	Available	vpc-0ab9222cd31d02273 App...	<input type="radio"/> Off	172.31.16.0/20	-

t a subnet

3. Create a route table and associate it with all three subnets, allowing internet-bound traffic by adding a route to the internet gateway.

Route tables (1/1) Info

Last updated 1 minute ago [Actions](#) [Create route table](#)

<input checked="" type="checkbox"/> Name	Route table ID	Explicit subnet associ...	Main	VPC
ApplicationRT	rtb-0de2905abfc36a848	subnet-0c77452e16c19d...	-	Yes vpc-0...

rtb-0de2905abfc36a848 / ApplicationRT

[Details](#) [Routes](#) [Subnet associations](#) [Edge associations](#) [Route propagation](#) [Tags](#)

Routes (2)

[Edit routes](#)

Destination	Target	Status	Propagated
0.0.0.0/0	igw-0c9ab6a110953517d	<input checked="" type="radio"/> Active	No
172.31.0.0/16	local	<input checked="" type="radio"/> Active	No

4. Go to EC2 and launch an instance named **Jenkins_server** within the configured VPC, ensuring it is set up for automation tasks without impacting the production environment.

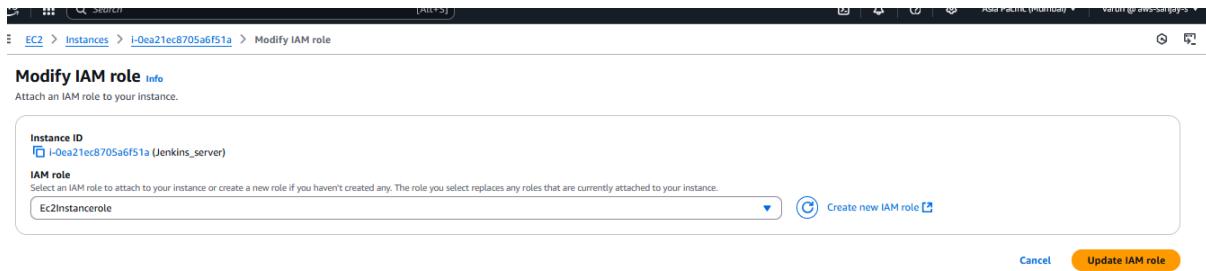
The screenshot shows the AWS EC2 Instances page. At the top, there's a search bar labeled "Find Instance by attribute or tag (case-sensitive)" and a dropdown menu set to "All states". Below the search bar, a table lists one instance: Jenkins_server (i-0e181b595ec11c955). The instance is running, has an instance type of t2.small, and is located in ap-south-1a with a public IPv4 DNS of ec2-13-233-106-252.ap... and a public IP of 13.233.106.252. The table includes columns for Name, Instance ID, Instance state, Instance type, Status check, Alarm status, Availability Zone, Public IPv4 DNS, and Public IP.

Below the table, the instance details for i-0e181b595ec11c955 (Jenkins_server) are shown. The "Details" tab is selected, followed by Status and alarms, Monitoring, Security, Networking, Storage, and Tags. Under the "Instance summary" section, it shows the Instance ID (i-0e181b595ec11c955), Public IPv4 address (13.233.106.252), Instance state (Running), and Private IPv4 address (172.31.35.149). It also shows the Public IPv4 DNS (ec2-13-233-106-252.ap-south-1.compute.amazonaws.com) and the Hostname type.

5. Create an IAM role with full access to **RDS, ECS, EC2 Instance Profile Image Builder for ECR containers**, ensuring it has the necessary permissions for seamless integration and automation.

The screenshot shows the AWS IAM Roles page. On the left sidebar, under "Access management", the "Roles" section is selected. In the main content area, a new role named "Ec2Instancerole" is being created. The "Summary" section shows the ARN (arn:aws:iam::590183945701:role/Ec2Instancerole), creation date (December 29, 2024, 01:17 (UTC+05:30)), and last activity (13 minutes ago). The "Permissions" tab is selected, showing three managed policies attached: AmazonECSTaskExecutionRole, AmazonRDSDataFullAccess, and EC2InstanceProfileForImageBuilderECRContainerRole. There are buttons for "Edit", "Delete", "Simulate", "Remove", and "Add permissions".

6. Assign the created IAM role to the **Jenkins EC2 instance**, allowing it to seamlessly access **RDS, ECS, EC2 Instance Profile Image Builder ECR containers** for smooth operation.



7. Install the required packages mentioned below,

a. Install Docker and docker-compose:

1. sudo apt update
2. sudo apt upgrade
3. sudo apt install -y apt-transport-https ca-certificates curl software-properties-common
4. curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg
5. echo "deb [arch=\$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu \$(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
6. sudo apt update
7. sudo apt install -y docker-ce docker-ce-cli containerd.io
8. sudo systemctl status docker
9. sudo apt install docker-compose

b. Install the AWS CLI:

1. sudo apt install aws-cli

OR

1. curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"
2. sudo apt update
3. sudo apt install unzip
4. unzip awscliv2.zip

5. sudo ./aws/install

c. Install the Jenkins:

1. sudo apt update

2. sudo apt install -y openjdk-17-jdk

3. curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null

4. echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/" | sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null

5. sudo apt update

6. sudo apt install -y jenkins

7. sudo systemctl enable --now jenkins

sudo systemctl status Jenkins

d. Give the full access to Jenkins user for using Docker:

1. sudo usermod -aG docker Jenkins

2. systemctl restart Jenkins

```
root@ip-172-31-7-152:~# sudo systemctl status docker
● docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib/systemd/system/docker.service; enabled; preset: enabled)
   Active: active (running) since Wed 2025-02-19 06:13:01 UTC; 15min ago
     TriggeredBy: * docker.socket
     Docs: https://docs.docker.com
      Main PID: 10052 (dockerd)
        Tasks: 9
       Memory: 21.6M (peak: 31.5M)
          CPU: 402ms
        CGroup: /system.slice/docker.service
               └─10052 /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock

Feb 19 06:13:01 ip-172-31-7-152 systemd[1]: Starting docker.service - Docker Application Container Engine...
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.265337343Z" level=info msg="Loading up"
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.265337343Z" level=info msg="OTEL tracing is not configured, using no-op tracer provider"
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.265590759Z" level=info msg="detected 127.0.0.53 nameserver, assuming systemd-resolved, so using resolv.conf: /run/resolvconf/nameservers"
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.461334641Z" level=info msg="Loading containers: start."
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.860844871Z" level=info msg="Loading containers: done."
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.886045448Z" level=info msg="Docker daemon" commit="4c9b3b0" containerd-snapshotter=false storage-driver=overlay2 vcpus=2
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.886169862Z" level=info msg="Daemon has completed initialization"
Feb 19 06:13:01 ip-172-31-7-152 dockerd[10052]: time="2025-02-19T06:13:01.936463997Z" level=info msg="API listen on /run/docker.sock"
Feb 19 06:13:01 ip-172-31-7-152 systemd[1]: Started docker.service - Docker Application Container Engine.
root@ip-172-31-7-152:~#
```

```

root@ip-172-31-7-152:~# sudo systemctl enable --now jenkins
root@ip-172-31-7-152:~# sudo systemctl status jenkins
● jenkins.service - Jenkins Continuous Integration Server
   Loaded: loaded (/usr/lib/systemd/system/jenkins.service; enabled; preset: enabled)
     Active: active (running) since Wed 2025-02-19 06:16:57 UTC; 10s ago
       PID: 14065 (Java)
      Tasks: 42 (limit: 1130)
     Memory: 292.4M (peak: 319.7M)
        CPU: 19.01s
       CGroup: /system.slice/jenkins.service
               └─14065 /usr/bin/java -Djava.awt.headless=true -jar /usr/share/java/jenkins.war --webroot=/var/cache/jenkins/war --httpPort=8080

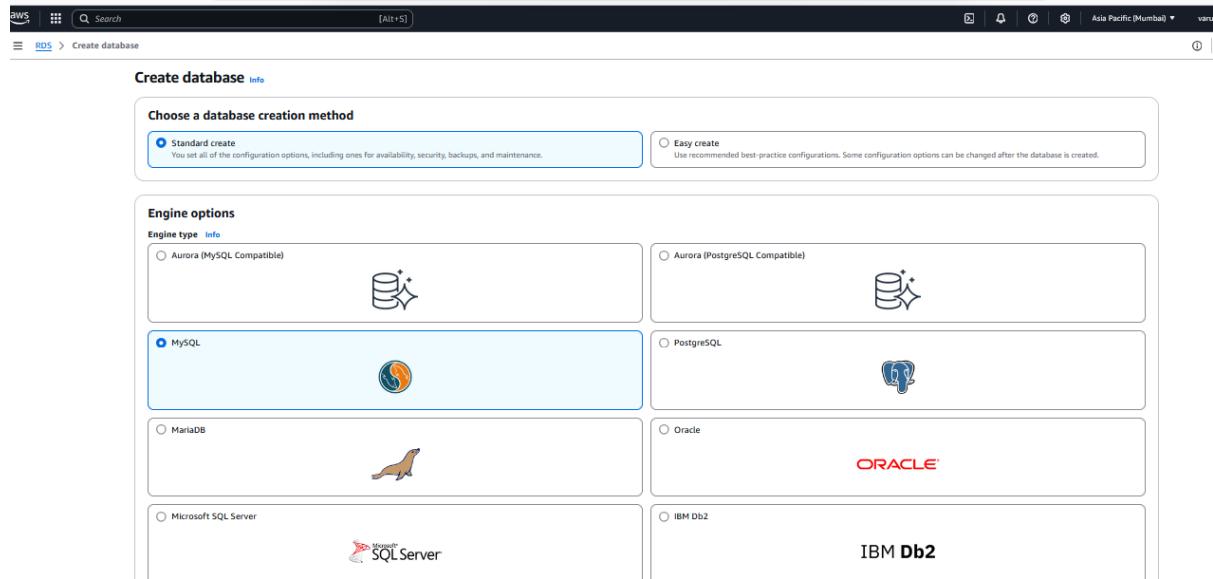
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: 7bb00c327e92411fab36e9509fd5ab9
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: This may also be found at: /var/lib/jenkins/secrets/initialAdminPassword
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: ****
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: ****
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: ****
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: 2025-02-19 06:16:57.680+0000 [id=31]      INFO  jenkins.InitReactorRunner$1#onAttained: Completed initialization
eb 19 06:16:48 ip-172-31-7-152 jenkins[14065]: 2025-02-19 06:16:57.680+0000 [id=31]      INFO  hudson.lifecycle.Lifecycle#onReady: Jenkins is fully up and running
eb 19 06:16:57 ip-172-31-7-152 systemd[1]: Started Jenkins Continuous Integration Server.
eb 19 06:16:59 ip-172-31-7-152 jenkins[14065]: 2025-02-19 06:16:59.490+0000 [id=46]      INFO  h.m.DownloadServices$Downloadable#load: Obtained the updated data file for hudson.ts
eb 19 06:16:59 ip-172-31-7-152 jenkins[14065]: 2025-02-19 06:16:59.491+0000 [id=46]      INFO  hudson.util.Retrigger$start: Performed the action check updates server successfully
root@ip-172-31-7-152:~# sudo usermod -aG docker jenkins
root@ip-172-31-7-152:~# su jenkins

```

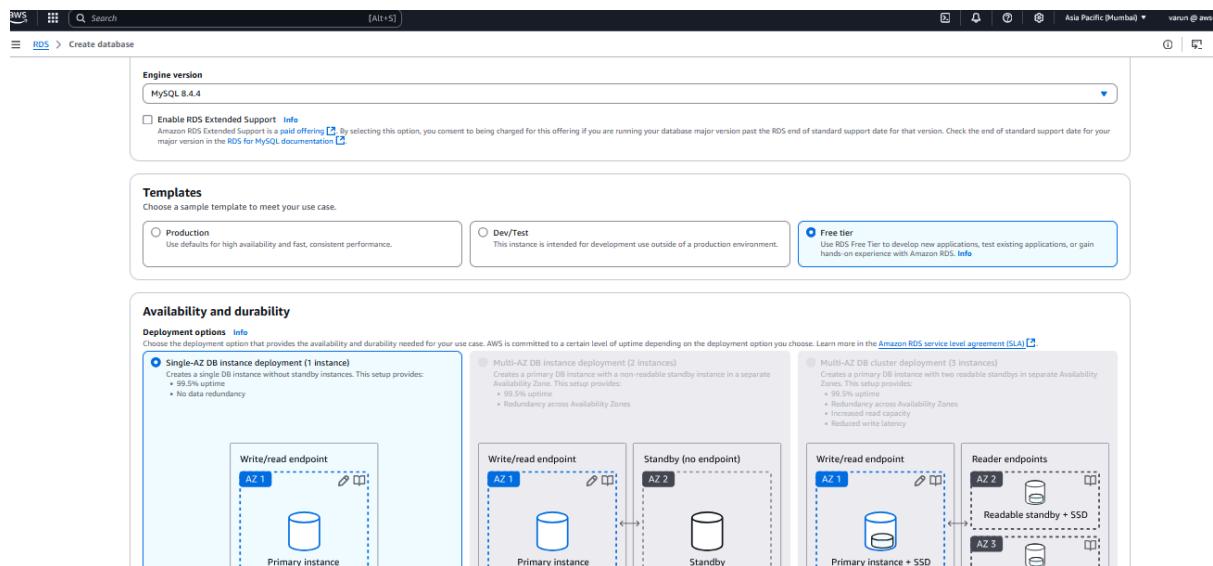
Module 2

Set up an RDS database, configure it appropriately, and create the necessary database and tables

1. Create an **RDS database** to store the application data. Use the **standard method** and select **MySQL** as the database engine based on the requirements.



2. Always choose the latest engine version or required for your and select the "Free Tier" template as we are doing for learning purposes.



3. Provide the database name and the credentials for logging into the RDS instance. Additionally, configure the CPU, memory, and storage according to the requirements.

The screenshot shows two sequential steps in the AWS RDS 'Create database' wizard:

- Settings Step:**
 - DB instance identifier:** python-web-application-db
 - Master username:** admin
 - Credentials Settings:** Self managed (selected)
 - Auto generate password:** Unselected
 - Master password:** [REDACTED]
 - Password strength:** Neutral
 - Confirm master password:** [REDACTED]
- Instance configuration Step:**
 - Instance configuration:** db.t4g.micro (selected)
 - Storage:** General Purpose SSD (gp2) - Allocated storage: 20 GiB
 - Connectivity:** Compute resource (selected)

4. For connectivity, select "**Don't connect to an EC2 instance**", as the goal is to connect the RDS database to **ECS** instead.

Connectivity Info

Compute resource
Choose whether to set up a connection to a compute resource for this database. Setting up a connection will automatically change connectivity settings so that the compute resource can connect to this database.

Don't connect to an EC2 compute resource
Don't set up a connection to a compute resource for this database. You can manually set up a connection to a compute resource later.

Connect to an EC2 compute resource
Set up a connection to an EC2 compute resource for this database.

Network type Info
To use dual-stack mode, make sure that you associate an IPv6 CIDR block with a subnet in the VPC you specify.

IPv4
Your resources can communicate only over the IPv4 addressing protocol.

Dual-stack mode
Your resources can communicate over IPv4, IPv6, or both.

Virtual private cloud (VPC) Info
Choose the VPC. The VPC defines the virtual networking environment for this DB instance.

Default VPC (vpc-0ab9222d31d02273)
3 Subnets, 3 Availability Zones
Only VPCs with a corresponding DB subnet group are listed.

After a database is created, you can't change its VPC.

DB subnet group Info
Choose the DB subnet group. The DB subnet group defines which subnets and IP ranges the DB instance can use in the VPC that you selected.

default-vpc-dab9222d31d02273
3 Subnets, 3 Availability Zones

Public access Info
Choose a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.

Yes
RDS assigns a public IP address to the database. Amazon EC2 instances and other resources outside of the VPC can connect to your database. Resources inside the VPC can also connect to the database. Choose one or more VPC security groups that specify which resources can connect to the database.

No
RDS doesn't assign a public IP address to the database. Only Amazon EC2 instances and other resources inside the VPC can connect to your database. Choose one or more VPC security groups that specify which resources can connect to the database.

VPC security group (firewall) Info
Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

Choose existing
Choose existing VPC security groups.

Create new
Create new VPC security group

5. Select the **automatic options**, keep the settings at their **default values**, and then click "**Create**" to set up the RDS database.

VPC security group (firewall) Info
Choose one or more VPC security groups to allow access to your database. Make sure that the security group rules allow the appropriate incoming traffic.

Choose existing
Choose existing VPC security groups.

Create new
Create new VPC security group

Existing VPC security groups
Choose one or more options
default

Availability Zone Info
No preference

Certificate authority - optional Info
Using a server certificate provides an extra layer of security by validating that the connection is being made to an Amazon database. It does so by checking the server certificate that is automatically installed on all databases that you provision.

rds-ca-rsa2048-g1 (default)
Expires: May 20, 2061
If you don't select a certificate authority, RDS chooses one for you.

Additional configuration

Tags - optional
A tag consists of a case-sensitive key-value pair.
No tags associated with the resource.
Add new tag
You can add up to 50 more tags.

Database authentication
Database authentication options Info
 Password authentication

Database authentication

Database authentication options Info

Password authentication
Authenticates using database passwords.

Password and IAM database authentication
Authenticates using the database password and user credentials through AWS IAM users and roles.

Password and Kerberos authentication
Choose a directory in which you want to allow authorized users to authenticate with this DB instance using Kerberos Authentication.

Monitoring Info
Choose monitoring tools for this database. Database Insights provides a combined view of Performance Insights and Enhanced Monitoring for your fleet of databases.

Database Insights - Advanced
• Retains 15 months of performance history
• Fleet-level monitoring
• Integration with CloudWatch Application Signals

Database Insights - Standard
• Retains 7 days of performance history, with the option to pay for the retention of up to 24 months of performance history

Database Insights pricing is separate from RDS monthly estimates. See [Amazon CloudWatch pricing](#).

Additional monitoring settings
Enhanced Monitoring, CloudWatch Logs and DevOps Guru

- Wait until the **RDS status** changes to "**Available**." Then, **note down the RDS endpoint**, as it will be required for configuring both the **application** and the **container**.

- Now, log in to the RDS from the server and set up the database along with its tables for the application.
 - `mysql -h <RDS_ENDPOINT> -u admin -p`
 - Give the password which given while creating the RDS

Give the following commands mentioned to below:

```
CREATE DATABASE IF NOT EXISTS django_crud;
```

```
USE django_crud;
```

```
CREATE TABLE IF NOT EXISTS app_register(
```

```
    id INT AUTO_INCREMENT PRIMARY KEY,
```

```
    firstname VARCHAR(50) NOT NULL,
```

```
    lastname VARCHAR(50) NOT NULL,
```

```
    email VARCHAR(50) NOT NULL,
```

```
    contact BIGINT
```

```
);
```

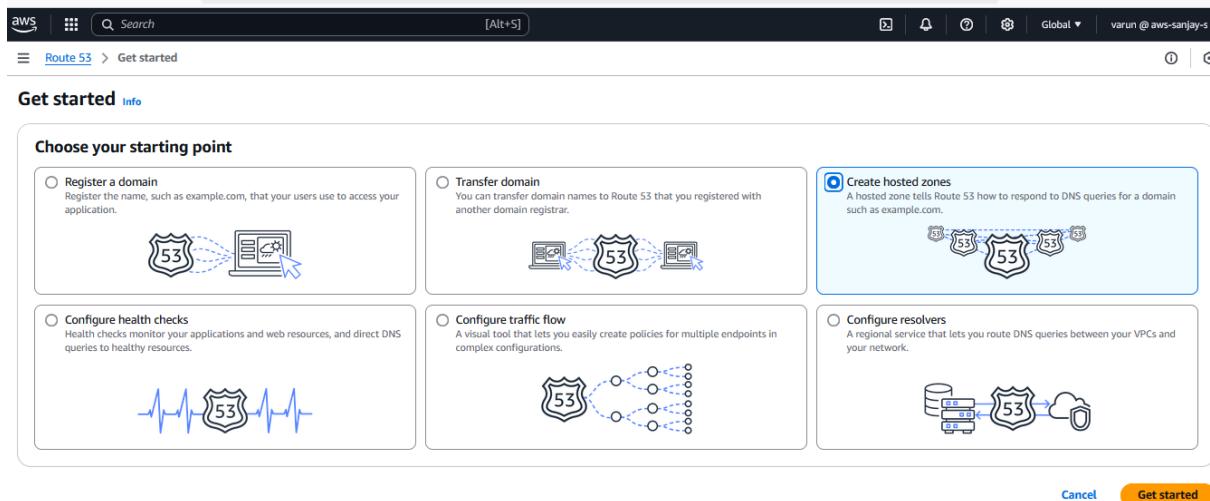
Module 03

Setting up a secure web application using Route 53, a Load Balancer, and ACM

1. Purchased or registered a domain (in this case, it was purchased through BigRock, a third-party provider).

The screenshot shows the BigRock domain management interface for the domain 'simha.in.net'. At the top, there is a green notification bar stating 'Name servers have been updated. It normally takes 24 - 72 hours to propagate'. Below this, the domain name 'simha.in.net' is displayed. Under 'DOMAIN REGISTRATION', it shows 'TITAN EMAIL (INDIA)' and an 'Order ID: 116252106'. The registration period is from '29th Nov, 2024' to '29th Nov, 2025', with a 'Renew' button and an 'Auto-renew' toggle switch. The 'NAME SERVERS & DNS' section contains 'Name Servers' (listing ns-1240.awsdns-27.org, ns-1637.awsdns-12.co.uk, ns-189.awsdns-23.com, ns-763.awsdns-31.net) and 'DNS Records'. The 'PREMIUM DNS' section offers 'Get optimized worldwide availability with our global Anycast DNS' and a 'BUY' button. The 'CONTACT DETAILS' section includes tabs for 'Domicient', 'Admin', 'Billing', 'Technical', and 'Privacy Protection'. On the left side of the interface, there are several decorative icons.

2. Create a hosted zone on Route 53 by providing the domain name you purchased and selecting the type as Public Hosted Zone.



Cancel

Get started

Create hosted zone

Hosted zone configuration

Domain name: simha.in.net

Description: The hosted zone is used for...

Type: Public hosted zone

Tags: Add tag

Cancel

Create hosted zone

3. After creating the hosted zone, copy the NS (Name Server) record values and paste them into the Name Server settings where you purchased the domain

Record Type	Name	Value	TTL
NS	simha.in.net	ns-798.awsdns-35.net. ns-1222.awsdns-24.org. ns-464.awsdns-58.com. ns-1931.awsdns-49.co.uk.	172800
SOA	simha.in.net	ns-798.awsdns-35.net. awsd...	900

4. Now, create a certificate in ACM for the region where the public instance is located. Choose the certificate type as Public and leave the remaining settings as default

Certificate type [Info](#)

ACM certificates can be used to establish secure communications access across the internet or within an internal network. Choose the type of certificate for ACM to provide.

Request a public certificate Request a public SSL/TLS certificate from Amazon. By default, public certificates are trusted by browsers and operating systems.

Request a private certificate No private CAs available for issuance.

Requesting a private certificate requires the creation of a private certificate authority (CA). To create a private CA, visit [AWS Private Certificate Authority](#).

[Cancel](#) [Next](#)

Domain names

Provide one or more domain names for your certificate.

Fully qualified domain name [Info](#)
simha.in.net

Add another name to this certificate You can add additional names to this certificate. For example, if you're requesting a certificate for "www.example.com", you might want to add the name "example.com" so that customers can reach your site by either name.

Validation method [Info](#)

Select a method for validating domain ownership.

DNS validation - recommended Choose this option if you are authorized to modify the DNS configuration for the domains in your certificate request.

Email validation Choose this option if you do not have permission or cannot obtain permission to modify the DNS configuration for the domains in your certificate request.

Key algorithm [Info](#)

Select an encryption algorithm. Some algorithms may not be supported by all AWS services.

RSA 2048 RSA is the most widely used key type.

ECDSA P 256 Equivalent in cryptographic strength to RSA 3072.

ECDSA P 384 Equivalent in cryptographic strength to RSA 7680.

[Add another name to this certificate](#) [Next Step](#)

5. After the certificate is created, copy the CNAME record and then add it to Route 53.

Successully requested certificate with ID [fa832423-64b8-497f-a7ed-2853df8696da](#)

A certificate request with a status of pending validation has been created. Further action is needed to complete the validation and approval of the certificate.

Domain	Status	Renewal status	Type	CNAME name
simha.in.net	Pending validation	-	CNAME	_2187974508a90af77e2ac5bf272aa343.simha.in.net

The screenshot shows the AWS Certificate Manager interface. At the top, it says "Successfully requested certificate with ID fa832423-64b8-497f-a7ed-2853df8696da". Below that, it says "A certificate request with a status of pending validation has been created. Further action is needed to complete the validation and approval of the certificate." There is a "View certificate" button. The main area is titled "Create DNS records in Amazon Route 53 (1/1)". It shows a table with one row: "simha.in.net" under "Domain", "Pending validation" under "Validation status", and "Yes" under "Is domain in Route 53?". There are "Cancel" and "Create records" buttons at the bottom.

The screenshot shows the AWS Route 53 Hosted Zones service. The left sidebar includes "Route 53", "Dashboard", "Hosted zones", "Health checks", "Profiles", "IP-based routing", "Traffic flow", "Domains", "Resolver", "VPCs", "Inbound endpoints", "Outbound endpoints", "Rules", and "Query logging". The main area shows the "simha.in.net" hosted zone details. Under "Records (3)", there are three entries:

Type	Name	Value	TTL
NS	simha.in.net	ns-798.awsdns-35.net, ns-1222.awsdns-24.org, ns-464.awsdns-58.com, ns-1931.awsdns-49.co.uk	172800
SOA	simha.in.net	ns-798.awsdns-35.net.awsd...	900
CNAME	_2187974...	_fc6b0eb3dc1a4b582207fdff...	300

Wait a few minutes for the validation to complete, and the status in the **ACM service** will change to "**Issued**".

6. Create a **Target Group** for the **Load Balancer**, selecting "**IP Address**" as the target type since we are using **serverless Fargate**.

The screenshot shows the AWS Lambda Target Groups creation wizard. Step 1: Specify group details. Step 2: Register targets. Under "Basic configuration", it says "Settings in this section can't be changed after the target group is created." Under "Choose a target type", the "IP addresses" option is selected. Other options include "Instances", "Lambda function", and "Application Load Balancer".

- Set the **protocol** to **TCP** and the **port** to **8000**, which is exposed/open in the target container.

Target group name: Python-Web-Application-LB-TG

Protocol : Port: TCP (8000)

IP address type: IPv4

VPC: ApplicationVPC (172.31.0.0/16)

Health checks: Health check protocol: TCP

- In the "Register Target" section, specify the **subnet** that needs to be targeted.

Step 1: Specify group details

Step 2: Register targets

IP addresses:

Step 1: Choose a network

Step 2: Specify IPs and define ports

Ports: 8000 (1-65535)

9. After creating the **Target Group**, set up an **Application Load Balancer (ALB)** to enable secure communication with the instance.
- Configure the **ALB as internet-facing**.
 - Select the **VPC** and its **three availability zones** where the container resides.

The screenshot shows the 'Network mapping' section of the AWS EC2 Load Balancers 'Create Application Load Balancer' page. It displays three selected Availability Zones (ap-south-1a, ap-south-1b, and ap-south-1c) and their corresponding subnets (ApplicationSubnet1, ApplicationSubnet2, and ApplicationSubnet3). Each subnet is associated with a specific VPC CIDR range.

10. In the **Listeners and Routing** section:

- Add **HTTP** listeners.
- Set the **default action** for both listeners to "**Forward to Target Group**".

The screenshot shows the 'Listeners and routing' section of the AWS EC2 Load Balancers 'Create Application Load Balancer' page. It lists a single listener named 'Listener HTTP:80' with the port set to 80. The default action is configured to forward requests to the 'Python-web-application-TG' target group using the HTTP protocol.

11. After creating the **Load Balancer**:

- Add another **HTTPS listener**.
- Choose the **default security policy**.
- For the **certificate**, select it from **ACM** and choose the certificate created earlier.
- Click "**Create**" to finalize the setup.

Add listener [Info](#)

Add a listener to your Application Load Balancer (ALB) to define how client requests and network traffic are routed within your application. Every listener is made up of a default action that's required and can only be edited. Additional rules can be added, edited and deleted from the listener.

Load balancer details: Python-Web-application-LB

Listener details: HTTPS:443

A listener checks for connection requests using the protocol and port that you configure. The default action and any additional rules that you create determine how the Application Load Balancer routes requests to its registered targets.

Listener configuration

The listener will be identified by the protocol and port.

Protocol
Used for connections from clients to the load balancer.

Port
The port on which the load balancer is listening for connections.

Default actions [Info](#)
The default action is used if no other rules apply. Choose the default action for traffic on this listener.

Authentication [Info](#)
Authentication requires IPv6 connectivity to authentication endpoints. [Learn more](#)

Use OpenID or Amazon Cognito
Include authentication using either OpenID Connect (OIDC) or Amazon Cognito.

Routing actions

Forward to target groups Redirect to URL Return fixed response

Forward to target group [Info](#)
Choose a target group and specify routing weight or [Create target group](#)

Forward to target group [Info](#)
Choose a target group and specify routing weight or [Create target group](#)

Target group
Python-Web-Application-LB-TG
Target type: IP, IPv4

Weight **Percent**

Add target group
You can add up to 4 more target groups.

Target group stickiness [Info](#)
Enables the load balancer to bind a user's session to a specific target group. To use stickiness the client must support cookies. If you want to bind a user's session to a specific target, turn on the Target Group attribute Stickiness.

Turn on target group stickiness

Secure listener settings [Info](#)

Security policy [Info](#)
Your load balancer uses a Secure Socket Layer (SSL) negotiation configuration called a security policy to manage SSL connections with clients. [Compare security policies](#)

Security category [Policy name](#)

Default SSL/TLS server certificate
The certificate ensures a client connects without SNI protocol, or if there are no matching certificates. You can source this certificate from AWS Certificate Manager (ACM), Amazon Identity and Access Management (IAM), or import a certificate. This certificate will automatically be added to your listener certificate list.

Certificate source

From ACM From IAM Import certificate

Certificate (from ACM)
The selected certificate will be applied as the default SSL/TLS server certificate for this load balancer's secure listeners.

[Request new ACM certificate](#)

12. Next, edit the HTTP listener:

- Change the **default action** to "Redirect to URL".
 - Set the **URL port** to **HTTPS (443)**.

This ensures that users accessing the application via **HTTP** are automatically redirected to **HTTPS** for a secure connection.

13. Now, in Route 53, create a record for HTTPS by setting the traffic routing value to the application Load Balancer and specifying its region, along with the Load Balancer's DNS name.

Module – 04

Set up ECR, ECS, task definition, and its service for containerized deployment

1. Create an **Elastic Container Registry (ECR)** on a **private repository**.

The screenshot shows two screenshots of the Amazon ECR interface.

Create private repository (Top Screenshot):

- General settings:** A repository name is specified as "590183945701.dkr.ecr.ap-south-1.amazonaws.com/python_web_application".
- Image tag immutability:** The "Mutable" option is selected, allowing image tags to be overwritten.
- Encryption:** AES-256 encryption is chosen.

Private repositories (Bottom Screenshot):

- A single repository named "python_web_application" is listed.
- Details for the repository:
 - Repository name: python_web_application
 - URI: 590183945701.dkr.ecr.ap-south-1.amazonaws.com/python_web_application
 - Created at: February 25, 2025, 09:32:41 (UTC+05:5)
 - Tag immutability: Mutable
 - Encryption type: AES-256

2. Now, create the **Elastic Container Service (ECS) cluster** and select **AWS Fargate** as the infrastructure since we are using a **serverless** approach.

Cluster configuration

Cluster name
python_web_application_cluster

Default namespace - optional
Select the namespace to specify a group of services that make up your application. You can overwrite this value at the service level.
python_web_application_cluster

Infrastructure Info Serverless

Your cluster is automatically configured for AWS Fargate (serverless) with two capacity providers. Add Amazon EC2 instances.

AWS Fargate (serverless)
Pay as you go. Use if you have tiny, batch, or burst workloads or for zero maintenance overhead. The cluster has Fargate and Fargate Spot capacity providers by default.

Amazon EC2 instances
Manual configurations. Use for large workloads with consistent resource demands.

External instances using ECS Anywhere can be registered after cluster creation is complete.

Monitoring - optional Info
CloudWatch Container Insights is a monitoring and troubleshooting solution for containerized applications and microservices.

Encryption - optional
Choose the KMS keys used by tasks running in this cluster to encrypt your storage.

Tags - optional Info
Tags help you to identify and organize your clusters.

Create

Clusters (1) Info

Last updated: February 25, 2025 at 12:02 (UTC+5:30)

Cluster	Services	Tasks	Container instances
python_web_application_cluster	0	No tasks running	0 EC2

Create cluster

3. After creating the ECR, proceed with creating the Task Definition:

- **Launch Type: Fargate** (as we are using serverless).
- **Operating System: Linux x86_64**.
- **Task Size:** Define the required **CPU and memory** for the containers.

4. Now, configure the container details:

- **Image URI:** Use the **ECR URI** created earlier.
- **Container Port:** **8000** (as exposed by the application).
- **Application Protocol:** **HTTP**.

5. Set up the **environment variables** to enable communication between the **container** and **RDS** or other containers:

- **DB_HOST** = *RDS Endpoint* (created earlier)
- **DB_NAME** = *django_crud* (database created on RDS)
- **DB_USER** = *Username set during RDS creation*
- **DB_PASSWORD** = *Password set during RDS creation*

6. Enable log collection to store application container logs in an Amazon CloudWatch Logs group.

7. Once the Task Definition is created, proceed with setting up the Service:

- Go to the **ECS Cluster**.
- Click on "**Create Service**" to begin the setup.

Task definition successfully created
python_app_application_task_definition:6 has been successfully created. You can use this task definition to deploy a service or run a task.

Last updated February 25, 2025 at 12:06 (UTC+5:30)

python_web_application_cluster

Cluster overview

ARN arn:aws:ecs:ap-south-1:590183945701:cluster/python_web_application_cluster	Status Active	CloudWatch monitoring Default	Registered container instances -
Services Draining -	Active	Tasks Pending -	Running

Services | Tasks | Infrastructure | Metrics | Scheduled tasks | Configuration | Tags

Services (0) Info

Create

- Set the **Capacity Provider** to **FARGATE** and set the **Base Value** to **1**, ensuring that at least **one task** is always running.

Create **Info**

Environment

Existing cluster
python_web_application_cluster

AWS Fargate

Compute configuration (advanced)

Compute options | **Info**
To ensure task distribution across your compute types, use appropriate compute options.

Capacity provider strategy | **Info**
Select either your cluster default capacity provider strategy or select the custom option to configure a different strategy.

Use cluster default
No default capacity provider strategy configured for this cluster.

Use custom (Advanced)

Capacity provider | **Info**
FARGATE

Base | **Info**
1

Weight | **Info**
1

Add capacity provider

Platform version | **Info**
Specify the platform version on which to run your service.

- Set the **Platform Version** to "**LATEST**", as the **ECR** stores only the **latest image** for deployment.

The screenshot shows the 'Create service' step in the AWS ECS wizard. On the left, there's a sidebar with navigation links like 'Amazon Elastic Container Service', 'Clusters', 'python_web_application_cluster', and 'Create service'. The main area is titled 'Deployment configuration'.

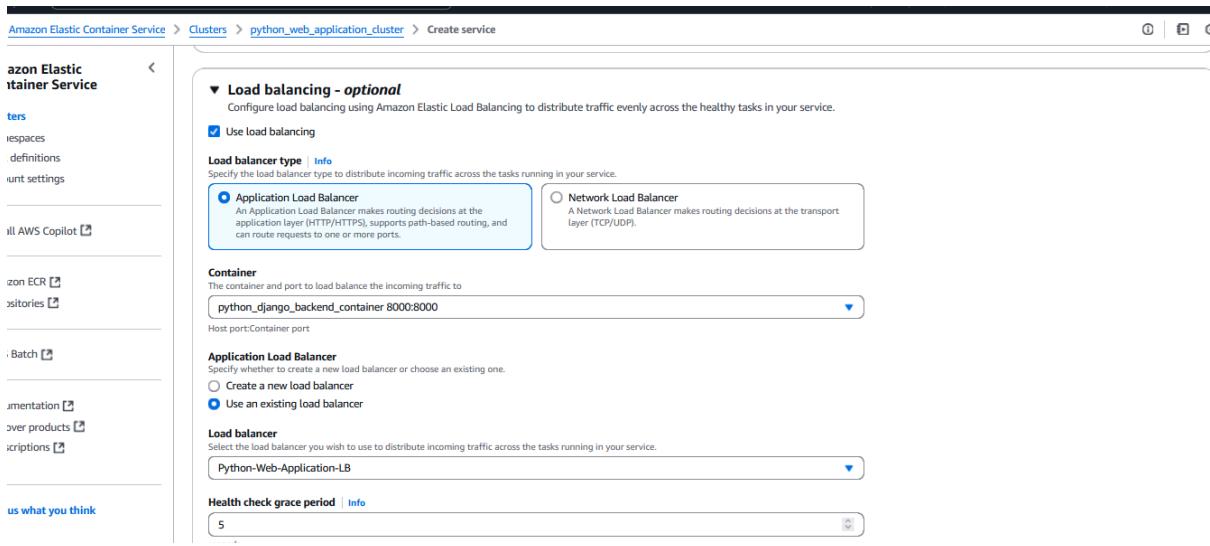
- Platform version:** Set to 'LATEST'.
- Application type:** Set to 'Service'.
- Task definition:** Set to 'Specify the revision manually' (checkbox checked), with 'python_app.application_task_definition' selected and '6 (LATEST)' chosen from the dropdown.
- Service name:** Set to 'python_web_application_service'.
- Service type:** Set to 'Replica'.
- Daemon:** Unchecked.

10. Keep the remaining settings as default and select the same VPC where the RDS and EC2 instance are set up. This ensures seamless connectivity without needing additional configurations.

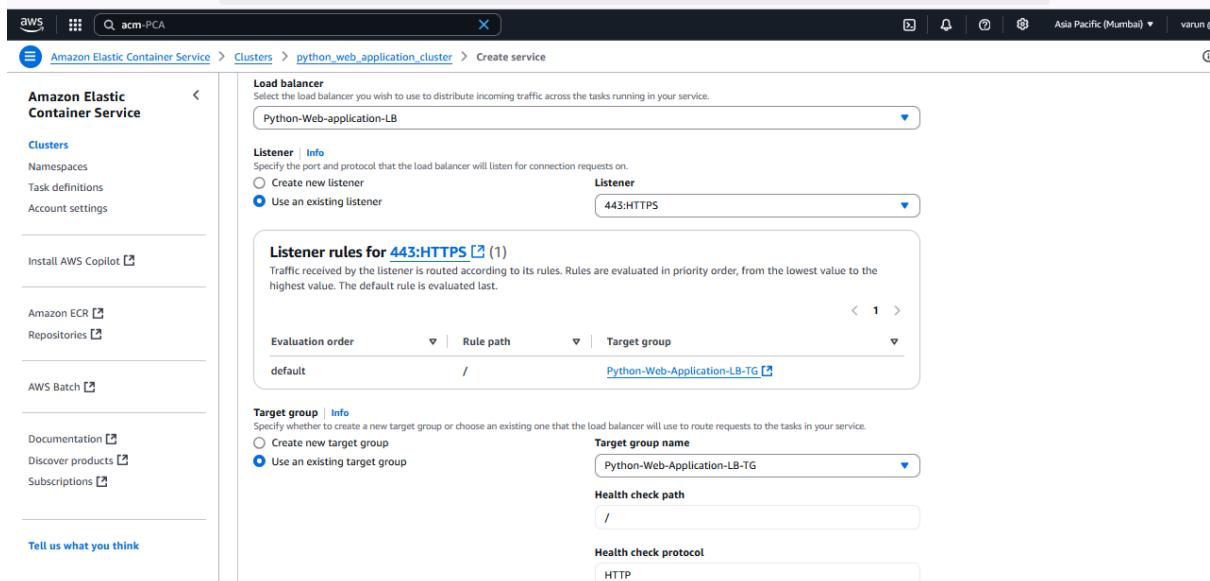
The screenshot shows the 'Networking' section of the 'Create service' wizard. It includes the following settings:

- VPC:** Set to 'vpc-0ab222cd31d02273' (ApplicationVPC | default).
- Subnets:** Three subnets are selected: 'subnet-0c77452e16c19ded6' (ApplicationSubnet1, ap-south-1b, 172.31.0.0/20), 'subnet-053819e25328c54e9' (ApplicationSubnet2, ap-south-1a, 172.31.32.0/20), and 'subnet-05efffab42df7892a' (ApplicationSubnet3, ap-south-1c, 172.31.16.0/20).
- Security group:** Set to 'sg-0f9603d774cded4b' (ECS_SG | ECS_SG).
- Public IP:** Set to 'Turned on'.

11. Now, select the Application Load Balancer (ALB) that was created earlier.



12. Here, select **HTTPS** instead of **HTTP** to ensure secure communication.



Note: Initially, the task will **fail** since no image is stored in **ECR**.

To prevent unnecessary failures:

- **Stop the running task** under the **Task** section in **ECS**.

The screenshot shows the AWS Elastic Container Service (ECS) Cluster Overview page. The cluster name is 'python_web_application_cluster'. The status is 'Active'. CloudWatch monitoring is set to 'Default'. There are no registered container instances. Services section shows 'Draining' with 1 active task. Tasks section shows 5 tasks: one provisioning and four running. One task is stopped.

Cluster overview

ARN	Status	CloudWatch monitoring	Registered container instances
arn:aws:ecs:ap-south-1:590183945701:cluster/python_web_application_cluster	Active	Default	-

Services

Draining	Active	Pending	Running
-	1	-	-

Tasks (5)

Task	Last status	Desired status	Task definition	Health status	Started by
7de7442de6b54a0db29b948dff5288c	Provisioning	Running	python_app_application_ta...	Unknown	ecs-svc/49281185964
11c3727d8c653461bb47c1fd9d1d1934	Stopped	Stopped	python_app_application_ta...	Unknown	ecs-svc/62360267874

Module – 05

Overview of the Web Application and Implementing Necessary Modifications

Before building and pushing the container, let's go through an **overview of the application**.

1. The **GitHub repository** contains a **Python web application** structured as follows:

```
|── docker-compose.yml  
|── backend  
|   |── App  
|   |   |── Templates  
|   |   |   |── edit.html  
|   |   |   |── insert.html  
|   |   |   |── show.html  
|   |   |── Migrations  
|   |   |── admin.py  
|   |   |── apps.py  
|   |   |── models.py  
|   |   |── tests.py  
|   |   |── urls.py  
|   |   |── views.py  
|   |── Basic  
|   |   |── settings.py  
|   |   |── urls.py  
|   |   |── wsgi.py  
|── Dockerfile  
|── manage.py
```

This structure includes:

- **Docker-related files** (docker-compose.yml, Dockerfile) for containerization.
- **Backend application** containing **Django models, views, templates, and configurations**.
- **Templates folder** for HTML views.
- **Settings and URLs** for application configuration.

Docker-compose.yml:

```
version: '3.8'

services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - DB_HOST= RDS-EndPoint( Which is created earlier)
      - DB_NAME=django_crud
      - DB_USER=Admin
      - DB_PASSWORD=Simha
    networks:
      - my_networks

networks:
  my_networks:
    driver: bridge
```

dockerfile (inside the bankend directory):

```
FROM python:3.9
WORKDIR /app
RUN apt-get update
COPY . /app
RUN pip install --no-cache-dir django mysqlclient
EXPOSE 8000
CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

2. Now, modify the necessary configurations in the **settings.py** file.
 - The application should be accessible from anywhere.
 - Update the ALLOWED_HOSTS in the settings.py file as follows:

```
ALLOWED_HOSTS = ['*', 'localhost', '127.0.0.1']
```

3. The application should only accept **CSRF tokens** from the specified domains.
Update the **CSRF_TRUSTED_ORIGINS** in the **settings.py** file:

```
CSRF_TRUSTED_ORIGINS = [
    "https://simha.in.net",
    "https://www.simha.in.net" # Add both with and without "www"
]
```

4. Update the **database connection settings** in the **settings.py** file to connect to the **RDS MySQL database**:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
        'NAME': os.environ.get('DB_NAME', 'django_crud'),
        'USER': os.environ.get('DB_USER', 'admin'),
```

```
'PASSWORD': os.environ.get('DB_PASSWORD', 'Simha54321'),  
'HOST': os.environ.get('DB_HOST', 'python-web-application-  
db.cxygosic8ugs.ap-south-1.rds.amazonaws.com'),  
'PORT': os.environ.get('DB_PORT', '3306'),  
,  
}  
}
```

Module – 06

Configure the Jenkins console and set up its pipeline code

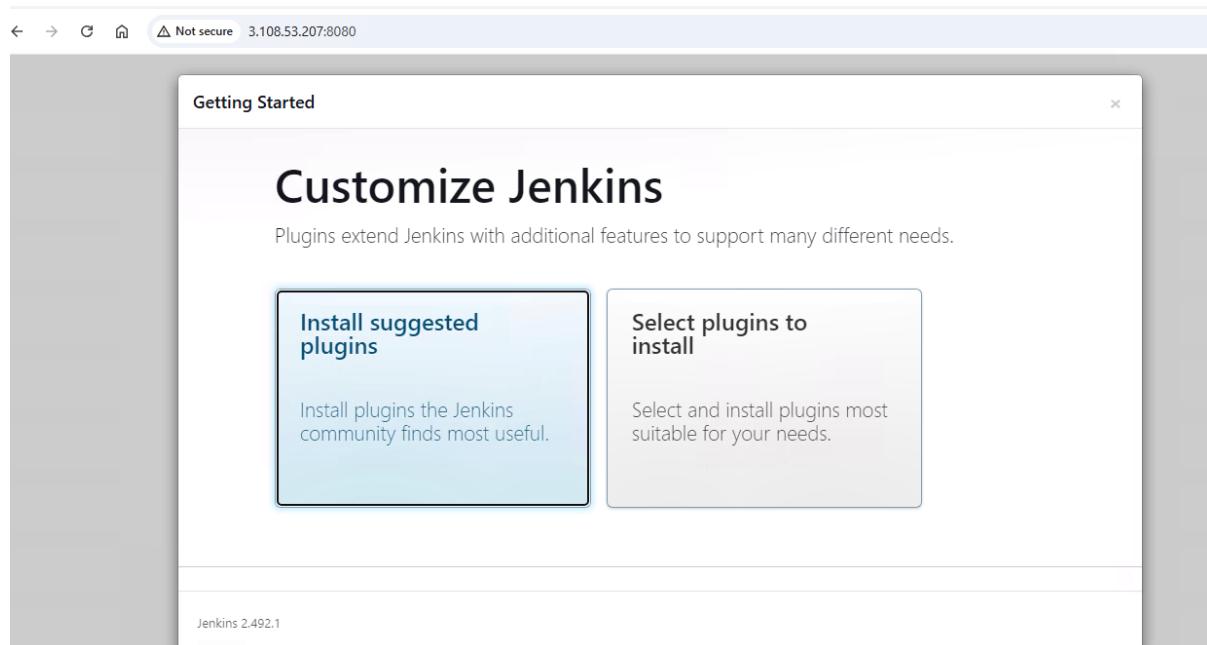
- Once the **Jenkins service** is running successfully on the server, access it using the **public IP address** with port **8080** in the browser.
 - Example:** <http://3.108.53.207:8080>
- Upon initialization, Jenkins will prompt for an **admin password**. Retrieve it using the following command:

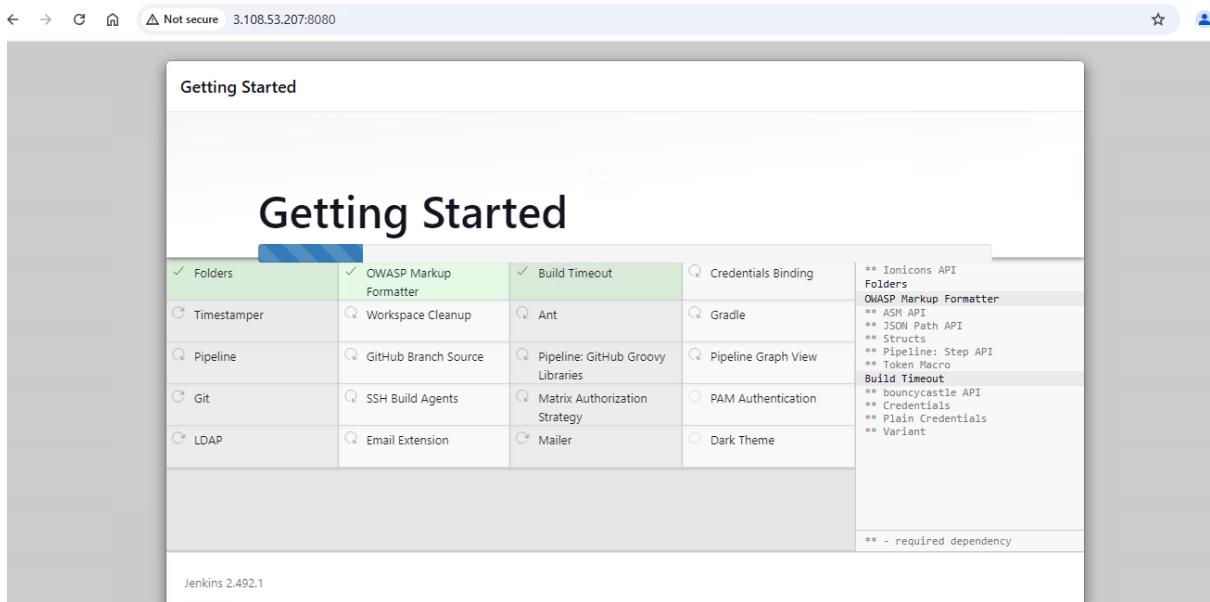
Retrieve Jenkins Initial Admin Password

```
sudo cat /var/lib/jenkins/secrets/initialAdminPassword
```

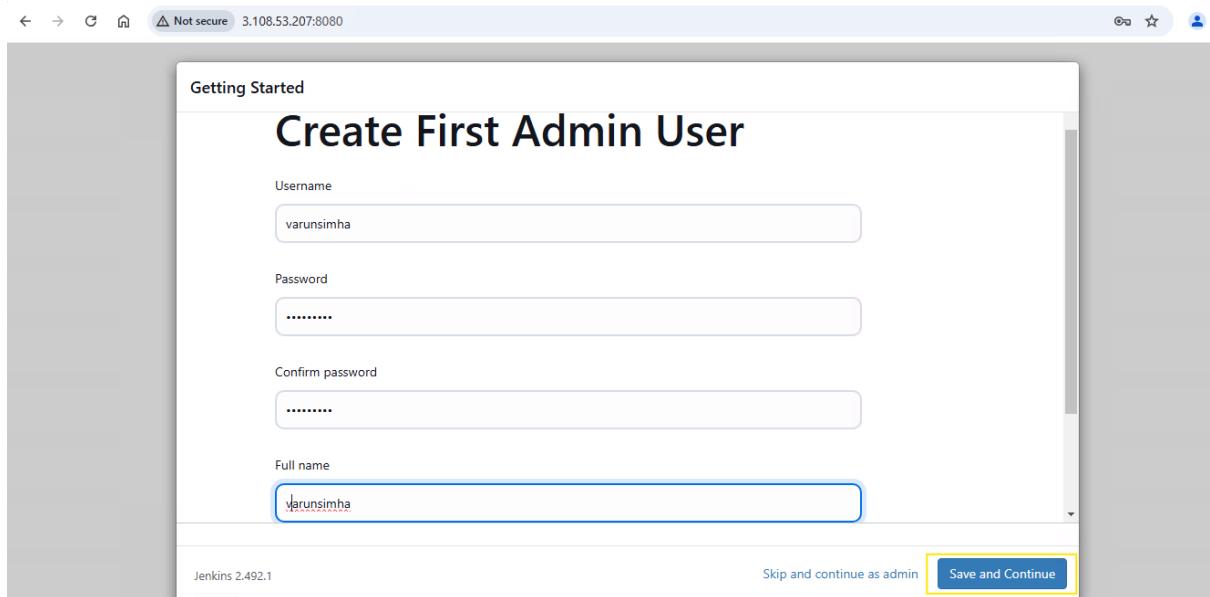
```
root@ip-172-31-7-152:/my_project# sudo cat /var/lib/jenkins/secrets/initialAdminPassword
7bb00c327e92411fab36e9509f05abd9
root@ip-172-31-7-152:/my_project#
```

- Next, install the **suggested plugins** by clicking on the "**Install suggested plugins**" option during the Jenkins setup.





- After the installation, enter the **required details** such as **username, password, full name, and email**, then click on "**Save and Continue**" to proceed.



- Since the GitHub repository is **private**, instead of hardcoding the token in the code, save it in **Jenkins Global Credentials**:
 - Go to **Jenkins Dashboard** → **Manage Jenkins** → **Manage Credentials**.
 - Under **Stores scoped to Jenkins**, click on **(global)** → **Add Credentials**.
 - Choose "**Secret Text**" as the credential type.
 - Enter the **GitHub Personal Access Token (PAT)** in the **Secret** field.
 - Provide an **ID** (e.g., github-token) for reference.
 - Click "**OK**" to save the credential.

This ensures secure authentication without exposing the token in the pipeline script.

The screenshot shows the Jenkins 'Global credentials (unrestricted)' configuration page. A red box highlights the 'Kind' dropdown set to 'Secret text'. Another red box highlights the 'Secret' field containing 'GITHUB_TOKEN'. A third red box highlights the 'ID' field also containing 'GITHUB_TOKEN'. The 'Description' field contains 'github token'. A blue 'Create' button is at the bottom.

6. Once the **Global Credentials** for the token are created, follow these steps to set up a **new Jenkins pipeline**:

- Go to **Jenkins Dashboard** and click on "**New Item**".
- Enter the **Item Name** (e.g., MyPipeline).
- Select "**Pipeline**" as the item type.
- Click "**OK**" to proceed.

The screenshot shows the Jenkins 'New Item' configuration page. A red box highlights the 'Item name' input field containing 'python_django_web_application_jenkins'. A red box highlights the 'Pipeline' item type option. A blue 'OK' button is at the bottom.

7. In the **Pipeline** section:

- Scroll down to the **Pipeline** configuration.
- Select "**Pipeline script**" as the definition.
- Enter the **pipeline script** to automate the workflow.
- Click "**Save**" to finalize the setup.

This script will define the build, test, and deployment steps for your Jenkins automation.

Pipeline script:

```
pipeline {  
    agent any  
    environment {  
        AWS_REGION = 'ap-south-1'  
        AWS_ACCOUNT_ID = '590183945701'  
        ECR_REPO =  
            "${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/python_web_application"  
        ECS_CLUSTER = 'python_web_application_cluster'  
        ECS_SERVICE = 'python_web_application_service'  
        TASK_DEFINITION = 'python_app_application_task_definition'  
    }  
    stages {  
        stage('Git clone') {  
            steps {  
                withCredentials([string(credentialsId: 'GITHUB_TOKEN', variable: 'github')]) {  
                    script {  
                        if (fileExists('Python-Web-Application-Jenkins/.git')) {  
                            dir('Python-Web-Application-Jenkins') {  
                                sh 'git reset --hard HEAD && git pull'  
                            }  
                        }  
                    else {  
                        sh 'git clone  
https://$github@github.com/varunsimha-MP/Python-Web-Application-Jenkins.git'  
                    }  
                }  
            }  
        }  
    }  
}
```

```

        }

    }

}

}

stage('Clean and build Docker Images on Jenkin server') {

    steps {

        dir('Python-Web-Application-Jenkins') {

            sh """
                docker-compose down --rmi all --remove-orphans
                docker-compose build
                """
            }

        }
    }
}

stage('Push Docker Images') {

    steps {

        dir('Python-Web-Application-Jenkins') {

            sh """
                docker login -u AWS -p \$(aws ecr get-login-password --region
                \$AWS_REGION) \$ECR_REPO

                #docker build -t \$ECR_REPO:latest .
                docker tag python_web_application:latest \$ECR_REPO:latest
                docker push \$ECR_REPO:latest
                """
            }

        }
    }
}

```

```

stage('Deploy to ECS') {
    steps {
        sh """
            aws ecs update-service --cluster ${ECS_CLUSTER} --service ${ECS_SERVICE}
            --force-new-deployment
            """
    }
}
}
}

```

Trigger builds remotely (e.g. from script)

Dashboard > python_django_web_application_jenkins > Configuration

Configure

Pipeline

Define your Pipeline using Groovy directly or pull it from source control.

General

Triggers

Pipeline

Advanced

Pipeline script

Use Groovy Sandbox [?](#)

```

1 pipeline {
2     agent any
3     environment {
4         AWS_REGION = 'ap-south-1'
5         AWS_ACCOUNT_ID = '590183945701'
6         ECR_REPO = "${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/python_web_application'
7         ECS_CLUSTER = 'python_web_application_cluster'
8         ECS_SERVICE = 'python_web_application_service'
9         TASK_DEFINITION = 'python_app_application_task_definition'
10    }
11    stages {
12        stage('Git clone') {
13            steps {
14                withCredentials([string(credentialsId: 'GITHUB_TOKEN', variable: 'github')]) {
15                    script {
16                        if (!fileExists('Python-Web-Application-Jenkins/.git')) {
17                            dir('Python-Web-Application-Jenkins') {
18                                sh 'git clone https://github.com/Pythontesting/Python-Web-Application-Jenkins.git'
19                            }
20                        }
21                    }
22                }
23            }
24        }
25    }
26}

```

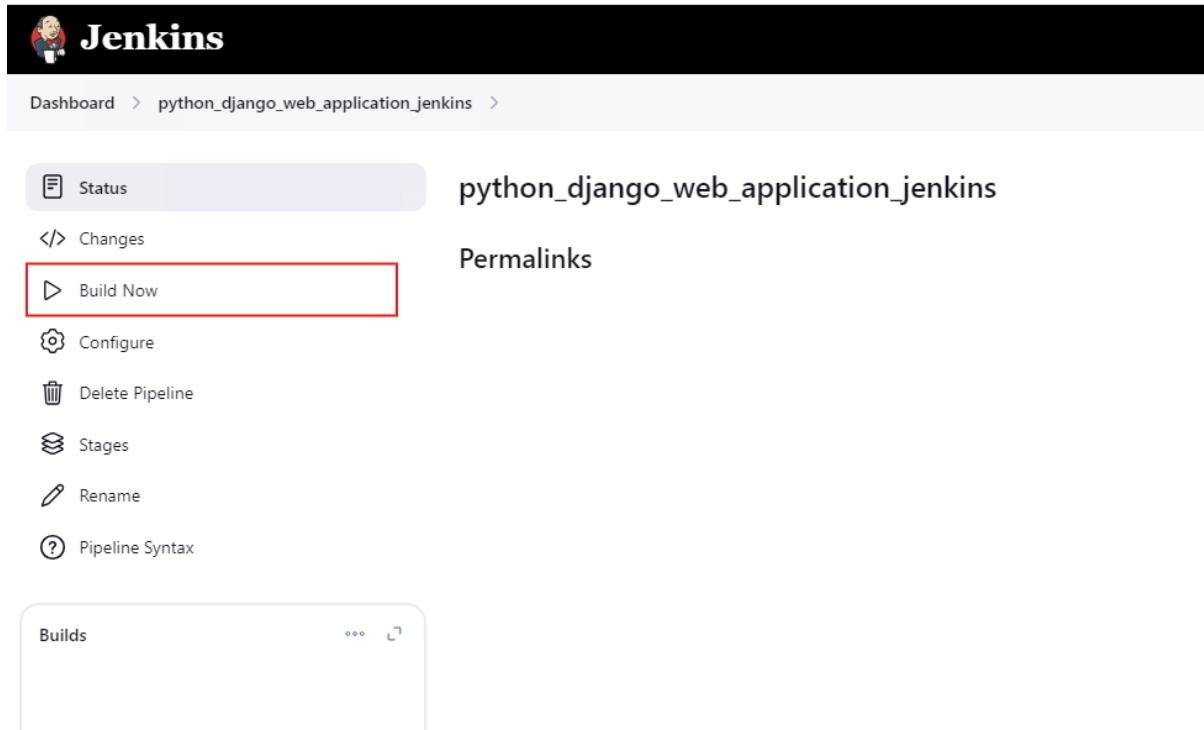
Module 7

Test and Validating the Setup

- Once the **Pipeline** item is ready:

- Click "**Build Now**" to trigger the pipeline.
- Monitor the build progress in the **Build History** section.
- Wait until the build completes successfully.

If any issues arise, check the **Console Output** for debugging. 



- After successfully **pushing/publishing** from the pipeline:

- Go to the **AWS ECS Console**.
- Navigate to your **Cluster**.
- Click on the **Services** tab and locate your service.
- Check the "**Run Task**" status.
- You should see a **running task** after the successful deployment.

If the task is not running, check the **logs** in **CloudWatch** or the **ECS Events** for debugging. 

Dashboard > python_django_web_application_jenkins

> Clusters > python_web_application_cluster > Tasks

3. Once the task is running successfully:

- Open your web browser.
- Enter the **DNS name** (simha.in.net).
- Access the application and navigate to the required page.
- Fill in the form with some values.
- Click **Submit** to test the functionality.

If any issues occur, check the **CloudWatch logs**, **RDS connection**, or **ECS task logs** for debugging.

PERSONAL INFORMATION

First name:

Last name:

Mail Id:

Phone Number: XXX-XXX-XXXX

Note: Only valid details are considered

Created by VSMP

ID	Firstname	Lastname	Email	Contact	Edit	Delete
3	varun	simha	varun@gmail.com	1234567890	<input type="button" value="Edit"/>	<input type="button" value="Delete"/>

4. While working on the application, the logs are stored in **CloudWatch Logs Group**, allowing us to monitor the application's activity.
 - For **troubleshooting container issues**, these logs can be helpful in:
 - Identifying errors or failures.
 - Monitoring application behavior and performance.
 - To access logs:
 - Go to **AWS CloudWatch Console**.
 - Navigate to **Logs Groups**.
 - Select the appropriate **log group** associated with your ECS container.
 - View and analyze the logs for any errors or warnings.

These logs will be crucial for **debugging** and ensuring smooth application operations.

CloudWatch > Log groups > /ecs/python_app_application_task_definition

/ecs/python_app_application_task_definition

Log group details

Log class Info Standard	Metric filters 0	Data protection
ARN arn:aws:log:ap-south-1:590183945701:log-group:/ecs/python_app_application_task_definition*	Subscription filters 0	Sensitive data count
Creation time 2 hours ago	Contributor Insights rules	Field indexes
Retention 1 day	KMS key ID	Transformer
Stored bytes -	Anomaly detection Configure	

Log streams (4)

Log stream	Last event time
ecs/python_django_backend_container/_05702e8d24e544bbbe8ac26ee6b49016	2025-02-26 05:09:35 (UTC)
ecs/python_django_backend_container/baf24ef68c924e52b4-f0bd7ab002a6d	2025-02-26 04:54:46 (UTC)
ecs/python_django_backend_container/081bd5a21ab4484b4d920f8acc558e7	2025-02-26 04:53:54 (UTC)
ecs/python_django_backend_container/cbbfa5ed3fe8d488619127d13a00d59a6	2025-02-26 04:38:32 (UTC)

Log groups > /ecs/python_app_application_task_definition > ecs/python_django_backend_container/_05702e8d24e544bbbe8ac26ee6b49016

2025-02-26T05:21:23.651Z [26/Feb/2025 05:21:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:21:23.677Z [26/Feb/2025 05:21:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:21:23.785Z [26/Feb/2025 05:21:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:21:53.084Z [26/Feb/2025 05:21:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:21:53.718Z [26/Feb/2025 05:21:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:21:53.719Z [26/Feb/2025 05:21:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:22:14.974Z Not Found: /images.png
2025-02-26T05:22:14.975Z [26/Feb/2025 05:22:14] "GET /images.png HTTP/1.1" 404 3155
2025-02-26T05:22:12.716Z [26/Feb/2025 05:22:12] "GET / HTTP/1.1" 200 1281
2025-02-26T05:22:23.723Z [26/Feb/2025 05:22:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:22:23.734Z [26/Feb/2025 05:22:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:22:53.729Z [26/Feb/2025 05:22:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:22:53.738Z [26/Feb/2025 05:22:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:22:53.764Z [26/Feb/2025 05:22:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:23:23.753Z [26/Feb/2025 05:23:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:23:23.768Z [26/Feb/2025 05:23:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:23:23.795Z [26/Feb/2025 05:23:23] "GET / HTTP/1.1" 200 1281
2025-02-26T05:23:33.118Z Not Found: /.git/config
2025-02-26T05:23:33.119Z [26/Feb/2025 05:23:11] "GET /.git/config HTTP/1.1" 404 3160
2025-02-26T05:23:49.707Z Not Found: /.git/config
2025-02-26T05:23:49.707Z [26/Feb/2025 05:23:49] "GET /.git/config HTTP/1.1" 404 3160
2025-02-26T05:23:53.769Z [26/Feb/2025 05:23:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:23:53.788Z [26/Feb/2025 05:23:53] "GET / HTTP/1.1" 200 1281
2025-02-26T05:23:53.808Z [26/Feb/2025 05:23:53] "GET / HTTP/1.1" 200 1281

No newer events at this moment. Auto retry paused. [Resume](#)

- Now, let's update the application and redeploy it using Jenkins:
 - We will modify the text "**Registration Data**" to "**Registered Data**" without causing any downtime.
 - Using **Git** from a local machine, we will:
 - Make the necessary changes in the application code.
 - Commit and push the changes to the **GitHub repository**.
 - Trigger the Jenkins pipeline to redeploy the updated application seamlessly.

```

MINGW64:/c/Users/varunsimha.premanara/Downloads/new/Python-Web-Application-Jenkins
varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ ls
backend/ docker-compose.yml

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ vi backend/app/templates/show.html

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ git add .

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ git commit -m "Updated the Registration Data name to Registered data on show.html page"
[master ec39d99] Updated the Registration Data name to Registered data on show.html page
Committer: Varunsimha Premanarasimha <varunsimha.premanarasimha@stratogent.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

git config --global --edit

After doing this, you may fix the identity used for this commit with:

git commit --amend --reset-author

1 file changed, 1 insertion(+), 1 deletion(-)

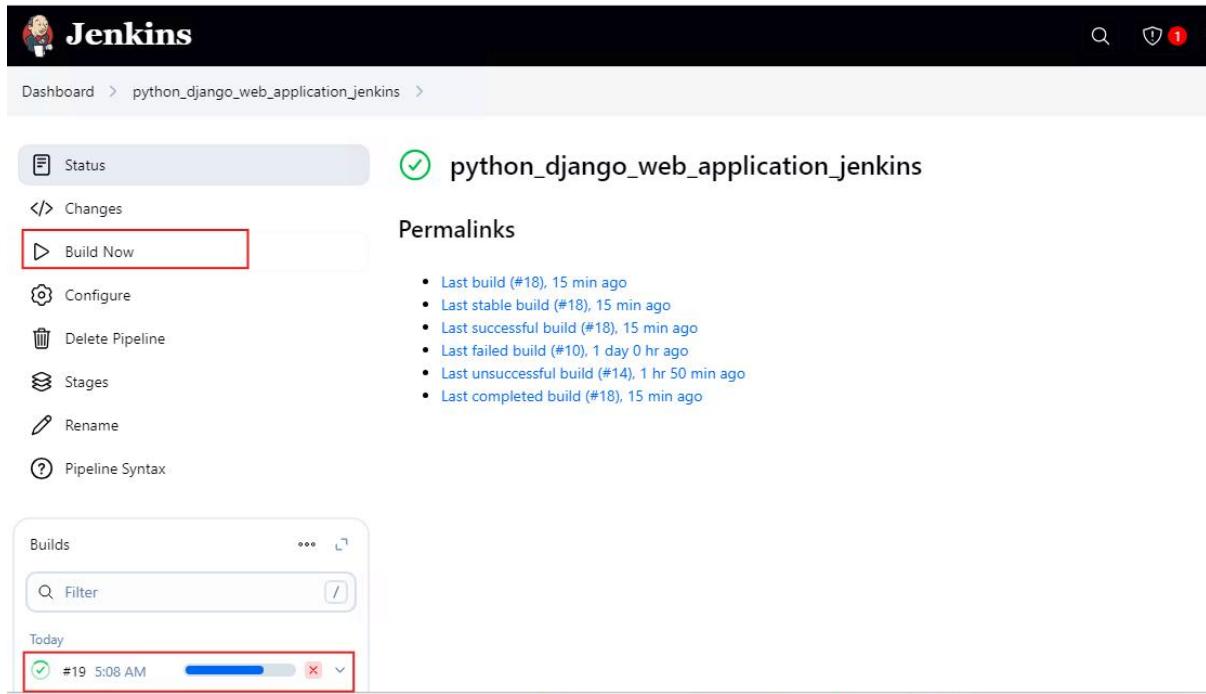
varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ git push
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 554 bytes | 138.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To https://github.com/varunsimha-MP/Python-Web-Application-Jenkins.git
  fff1d42..ec39d99 master -> master

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ ...

```

6. Once the changes are pushed to the **GitHub repository**, we will trigger the deployment from **Jenkins** by simply clicking the "**Build Now**" button.

There is no need to modify the pipeline script, as it will automatically fetch and deploy the updated code.



7. After the pipeline successfully completes, navigate to the **tasks** under the **ECS cluster**. You will see a new container being deployed, and its status will indicate that it is in the **provisioning** phase.

The screenshot shows the AWS ECS Cluster Overview page. The cluster name is 'python_web_application_cluster'. The 'Tasks' tab is active, displaying 15 tasks. One task is shown as 'Provisioning'.

ID	Status
05702e8d24e5448bbe8ac26ee6b49016	Provisioning
081bd5a2bab848d4bd920f8acec558e7	Running
1be1d7eb92554481b70691246576301	Stopped CannotPullContainerError: pull image manifest has been retried 1 time(s): failed to resolve ref 590183945701.dkr.ecr.ap-south-1.amazonaws.com/python_web:latest

8. Once the new container is up and running, the **Load Balancer** will automatically switch traffic to it, ensuring the updated changes are reflected seamlessly without any downtime.

The screenshot shows the AWS ECS Cluster Overview page. The cluster name is 'python_web_application_cluster'. The 'Tasks' tab is active, displaying 15 tasks. Two tasks are highlighted as 'Running'.

ID	Status
05702e8d24e5448bbe8ac26ee6b49016	Running
081bd5a2bab848d4bd920f8acec558e7	Running

9. Once the new container goes live, the previous container will automatically stop, ensuring a smooth and seamless deployment process.

Task	Last status	Desired state	Task state	Health state	Started by	Started at
05703d8234a444600ba26a0d49016	Running	Running	python...	Unknown	ecs-ecr/6445782324...	1 minute ago
081bd5a2ba844600ba26a0d49016	Deactivating	Stopped	python...	Unknown	ecs-ecr/64654810670...	17 minutes ago
1bc17e07c25254461077089134674261	Stopped	Stopped	python...	Unknown	ecs-ecr/76991195190...	-
5ed567bcac19427b023294a4976d80	Stopped	Stopped	python...	Unknown	ecs-ecr/49753978554...	-
5ed57795bb77147a011136c7367a198	Stopped	Stopped	python...	Unknown	ecs-ecr/76991195190...	-

5. Overall, of this project:

This architecture diagram represents a **Fully Automated Scalable CI/CD Pipeline for Serverless Django on AWS**, integrating multiple AWS services for scalability, security, and automation. Here's a breakdown of the components:

1. User Access via Route 53

- End-users access the application using the domain simha.in.net, managed by AWS Route 53.
- The traffic is routed through a Load Balancer, which:
 - Redirects HTTP (port 80) to HTTPS (port 443) for security.
 - Dynamically routes traffic to target groups on port 8000.

2. AWS Fargate for Containerized Django Application

- The application is containerized using Docker and deployed on AWS Fargate (serverless container orchestration via ECS).
- The Elastic Container Registry (ECR) stores the Docker images for deployment.
- The Django application inside the container runs on port 8000.

3. Database - Amazon RDS

- The Django application connects to an Amazon RDS instance as its backend database.

4. Logging and Monitoring

- Amazon CloudWatch monitors and logs container activities for debugging and performance tracking.

5. CI/CD Pipeline using Jenkins

- Jenkins Server runs on an EC2 instance to automate the deployment process.

- Developers write and push code from VS Code to GitHub.
- Jenkins pulls code from GitHub, builds the Docker image, and pushes it to ECR.
- The new image is deployed on Fargate automatically.

Key Takeaways:

- **Scalable Django Deployment:** Using **Fargate** to avoid managing EC2 instances manually.
- **Security:** HTTPS enforcement and centralized authentication.
- **Automation:** Jenkins handles CI/CD, reducing manual deployments.
- **Monitoring:** Logs and performance tracking via CloudWatch.

This setup ensures an efficient, scalable, and automated deployment pipeline for Django applications on AWS. 