UNIT – III

Python for Cloud

Outline

- Python for Amazon Web Services
- Python for Google Cloud
- Python for Windows Azure
- Python for MapReduce
- Python Packages of Interest
- Python Web Application Framework - Django
- Development with Django

Amazon EC2 – Python Example

- Boto is a Python package that provides interfaces to Amazon Web Services (AWS)

  - In this example, a connection to EC2 service is first established by calling boto.ec2.connect_to_region.
  - The EC2 region, AWS access key and AWS secret key are passed to this function. After connecting to EC2 , a new instance is launched using the conn.run_instances function.
  - The AMI-ID, instance type, EC2 key handle and security group are passed to this function.

```
#Python program for launching an EC2 instance
import boto.ec2
from time import sleep
ACCESS_KEY="<enter access key>"
SECRET_KEY="<enter secret key>"

REGION="us-east-1"  AMI_ID = "ami-
d0f89fb9"
EC2_KEY_HANDLE = "<enter key handle>"
INSTANCE_TYPE="t1.micro"
SECGROUP_HANDLE="default"

conn = boto.ec2.connect_to_region(REGION, aws_access_key_id=ACCESS_KEY,
                                  aws_secret_access_key=SECRET_KEY)

reservation = conn.run_instances(image_id=AMI_ID, key_name=EC2_KEY_HANDLE,
                                 instance_type=INSTANCE_TYPE,
                                 security_groups = [ SECGROUP_HANDLE, ] )
```

Amazon AutoScaling – Python Example

- **AutoScaling Service**
  - A connection to AutoScaling service is first established by calling **boto.ec2.autoscale.connect_to_region** function.
- **Launch Configuration**
  - After connecting to AutoScaling service, a new launch configuration is created by calling **conn.create_launch_configuration**. Launch configuration contains instructions on how to launch new instances including the AMI-ID, instance type, security groups, etc.
- **AutoScaling Group**
  - After creating a launch configuration, it is then associated with a new AutoScaling group. AutoScaling group is created by calling **conn.create_auto_scaling_group**. The settings for AutoScaling group such as the maximum and minimum number of instances in the group, the launch configuration, availability zones, optional load balancer to use with the group, etc.

```
#Python program for creating an AutoScaling group (code excerpt)
import boto.ec2.autoscale
:
print "Connecting to Autoscaling Service"
conn = boto.ec2.autoscale.connect_to_region(REGION,
   aws_access_key_id=ACCESS_KEY,
   aws_secret_access_key=SECRET_KEY)

print "Creating launch configuration"

lc = LaunchConfiguration(name='My-Launch-Config-2',
                         image_id=AMI_ID,
                         key_name=EC2_KEY_HANDLE,
                         instance_type=INSTANCE_TYPE,
                         security_groups = [ SECGROUP_HANDLE, ])
conn.create_launch_configuration(lc)

print "Creating auto-scaling group"

ag = AutoScalingGroup(group_name='My-Group',
             availability_zones=['us-east-1b'],
             launch_config=lc, min_size=1, max_size=2,
             connection=conn)
conn.create_auto_scaling_group(ag)
```

Amazon AutoScaling – Python Example

- **AutoScaling Policies**
  - After creating an AutoScaling group, the policies for scaling up and scaling down are defined.
  - In this example, a scale up policy with adjustment type **ChangeInCapacity** and **scaling_ad justment = 1** is defined.
  - Similarly a scale down policy with adjustment type **ChangeInCapacity** and **scaling_ad justment = -1** is defined.

```
#Creating auto-scaling policies
scale_up_policy = ScalingPolicy(name='scale_up',
                    adjustment_type='ChangeInCapacity', as_name='My-Group',
                    scaling_adjustment=1,  cooldown=180)

scale_down_policy = ScalingPolicy(name='scale_down',
                    adjustment_type='ChangeInCapacity', as_name='My-Group',
                    scaling_adjustment=-1,  cooldown=180)

conn.create_scaling_policy(scale_up_policy)
conn.create_scaling_policy(scale_down_policy)
```

Amazon AutoScaling – Python Example

- **CloudWatch Alarms**
  - With the scaling policies defined, the next step is to create Amazon CloudWatch alarms that trigger these policies.
  - The scale up alarm is defined using the CPU Utilization metric with the Average statistic and threshold greater 70% for a period of 60 sec. The scale up policy created previously is associated with this alarm. This alarm is triggered when the average CPU utilization of the instances in the group becomes greater than 70% for more than 60 seconds.
  - The scale down alarm is defined in a similar manner with a threshold less than 50%.

```
#Connecting to CloudWatch
cloudwatch = boto.ec2.cloudwatch.connect_to_region(REGION,
                        aws_access_key_id=ACCESS_KEY,
                        aws_secret_access_key=SECRET_KEY)
alarm_dimensions = {"AutoScalingGroupName": 'My-Group'}

#Creating scale-up alarm
scale_up_alarm = MetricAlarm(  name='scale_up_on_cpu',
          namespace='AWS/EC2', metric='CPUUtilization',
          statistic='Average',  comparison='>', threshold='70',
          period='60', evaluation_periods=2,
          alarm_actions=[scale_up_policy.policy_arn],
          dimensions=alarm_dimensions)
cloudwatch.create_alarm(scale_up_alarm)

#Creating scale-down alarm
scale_down_alarm = MetricAlarm(  name='scale_down_on_cpu',
          namespace='AWS/EC2', metric='CPUUtilization',
          statistic='Average',  comparison='<', threshold='40',
          period='60', evaluation_periods=2,
          alarm_actions=[scale_down_policy.policy_arn],
          dimensions=alarm_dimensions)
cloudwatch.create_alarm(scale_down_alarm)
```

Amazon S3 – Python Example

- In this example, a connection to S3 service is first established by calling **boto.connect_s3** function.
- The **upload_to_s3_bucket_path** function uploads the file to the S3 bucket specified at the specified path.

```
# Python program for uploading a file to an S3 bucket
import boto.s3

conn = boto.connect_s3(aws_access_key_id='<enter>',
   aws_secret_access_key='<enter>')

def percent_cb(complete, total): print ('.')

def upload_to_s3_bucket_path(bucketname, path, filename):
        mybucket = conn.get_bucket(bucketname)
        fullkeyname=os.path.join(path,filename)
        key = mybucket.new_key(fullkeyname)
        key.set_contents_from_filename(filename, cb=percent_cb, num_cb=10)
```

Amazon RDS – Python Example

- In this example, a connection to **RDS(Relational Database Service)**service is first established by calling **boto.rds.connect_to_region** function.
- The RDS region, AWS access key and AWS secret key are passed to this function.
- After connecting to RDS service, the **conn.create_dbinstance** function is called to launch a new RDS instance.
- The input parameters to this function include the instance ID, database size, instance type, database username, database password, database port, database engine (e.g. MySQL5.1), database name, security groups, etc.

```
#Python program for launching an RDS instance (excerpt)
import boto.rds

ACCESS_KEY="<enter>"
SECRET_KEY="<enter>"
REGION="us-east-1"
INSTANCE_TYPE="db.t1.micro"
ID = "MySQL-db-instance-3"
USERNAME = 'root'
PASSWORD = 'password'
DB_PORT = 3306
DB_SIZE = 5
DB_ENGINE = 'MySQL5.1'
DB_NAME = 'mytestdb'
SECGROUP_HANDLE="default"

#Connecting to RDS
conn = boto.rds.connect_to_region(REGION,
  aws_access_key_id=ACCESS_KEY,
  aws_secret_access_key=SECRET_KEY)


#Creating an RDS instance
db = conn.create_dbinstance(ID, DB_SIZE, INSTANCE_TYPE,
USERNAME, PASSWORD, port=DB_PORT, engine=DB_ENGINE,
db_name=DB_NAME, security_groups = [ SECGROUP_HANDLE, ] )
```

Amazon DynamoDB – Python Example

- In this example, a connection to DynamoDB service is first established by calling **boto.dynamodb.connect_to_region**.
- After connecting to DynamoDB service, a schema for the new table is created by calling **conn.create_schema**.
- The schema includes the **hash key** and **range key names** and **types**.
- A DynamoDB table is then created by calling **conn.create_table** function with the table schema, read units and write units as input parameters.

```
# Python program for creating a DynamoDB table (excerpt)
import boto.dynamodb

ACCESS_KEY="<enter>"
SECRET_KEY="<enter>"
REGION="us-east-1"

#Connecting to DynamoDB
conn = boto.dynamodb.connect_to_region(REGION,
  aws_access_key_id=ACCESS_KEY,
  aws_secret_access_key=SECRET_KEY)

table_schema = conn.create_schema(
      hash_key_name='msgid',
      hash_key_proto_value=str,
      range_key_name='date',
      range_key_proto_value=str
  )

#Creating table with schema
table = conn.create_table(
      name='my-test-table',
      schema=table_schema,
      read_units=1,
      write_units=1
  )
```

Google Compute Engine – Python Example

- This example uses the **OAuth 2.0** scope (https://www.googleapis.com/auth/compute) and credentials in the credentials file to request a refresh and access token, which is then stored in the **oauth2.dat** file.
- After completing the OAuth authorization, an instance of the Google Compute Engine service is obtained.
- To launch a new instance the **instances().insert** method of the Google Compute Engine API is used.
- The request body to this method contains the properties such as **instance name, machine type, zone, network interfaces, etc.,** specified in JSON format.

```
# Python program for launching a GCE instance (excerpt)
API_VERSION = 'v1beta15'
GCE_SCOPE = 'https://www.googleapis.com/auth/compute'
GCE_URL = 'https://www.googleapis.com/compute/%s/projects/' %
(API_VERSION) DEFAULT_ZONE = 'us-central1-b'
CLIENT_SECRETS =
'client_secrets.json'
OAUTH2_STORAGE = 'oauth2.dat'

def main():
 #OAuth 2.0 authorization.
 flow = flow_from_clientsecrets(CLIENT_SECRETS,
 scope=GCE_SCOPE) storage = Storage(OAUTH2_STORAGE)
 credentials = storage.get()

 if credentials is None or credentials.invalid:
  credentials = run(flow, storage)
 http = httplib2.Http()
 auth_http = credentials.authorize(http)

 gce_service = build('compute', API_VERSION)

# Create the instance
 request = gce_service.instances().insert(project=PROJECT_ID, body=instance,
                                           zone=DEFAULT_ZONE)
 response = request.execute(auth_http)
```

Google Cloud Storage – Python Example

- This example uses the **Oauth 2.0** scope (https://[www.googleapis.com/auth/devstora](http://www.googleapis.com/auth/devstorage.full_control) ge.full_control) and credentials in the credentials file to request a refresh and access token, which is then stored in the **oauth2.dat** file.
- After completing the OAuth authorization, an instance of the Google Cloud Storage service is obtained.
- To upload a file the **objects().insert** method of the Google Cloud Storage API is used.
- The request to this method contains the bucket name, file name and media body containing the MediaIoBaseUpload object created from the file contents.

```
# Python program for uploading a file to GCS (excerpt)

def main():
 #OAuth 2.0 authorization.
 flow = flow_from_clientsecrets(CLIENT_SECRETS,
 scope=GS_SCOPE) storage = Storage(OAUTH2_STORAGE)
 credentials = storage.get()

 if credentials is None or credentials.invalid:
  credentials = run(flow,
 storage) http = httplib2.Http()
 auth_http = credentials.authorize(http)

 gs_service = build('storage', API_VERSION, http=auth_http)

# Upload file
 fp= open(FILENAME,'r')
 fh = io.BytesIO(fp.read())
 media = MediaIoBaseUpload(fh, FILE_TYPE)
      request = gs_service.objects().insert(bucket=BUCKET,
                              name=FILENAME,
                                     media_body=media)
 response = request.execute()
```

Google Cloud SQL – Python Example

- This example uses the **OAuth 2.0** scope (https://www.googleapis.com/auth/compute) and credentials in the credentials file to request a refresh and access token, which is then stored in the **oauth2.dat** file.
- After completing the OAuth authorization, an instance of the Google Cloud SQL service is obtained.
- To launch a new instance the **instances().insert** method of the Google Cloud SQL API is used.
- The request body of this method contains properties such as **instance, project, tier, pricingPlan and replicationType**.

 # Python program for launching a Google Cloud SQL instance (excerpt)

```
def main():
 #OAuth 2.0 authorization.
 flow = flow_from_clientsecrets(CLIENT_SECRETS,
 scope=GS_SCOPE) storage = Storage(OAUTH2_STORAGE)
 credentials = storage.get()

 if credentials is None or credentials.invalid:
   credentials = run(flow,
 storage) http =
 httplib2.Http()
 auth_http = credentials.authorize(http)

 gcs_service = build('sqladmin', API_VERSION, http=auth_http)

 # Define request body
instance={"instance":
"mydb", "project":
"bahgacloud",  "settings":{
  "tier":  "D0",
  "pricingPlan":
  "PER_USE",
  "replicationType": "SYNCHRONOUS"}}

# Create the instance
 request = gcs_service.instances().insert(project=PROJECT_ID,
 body=instance) response = request.execute()
```

Azure Virtual Machines – Python Example

- To create a virtual machine, a cloud service is first created.
- Virtual machine is created using the **create_virtual_machine_deployment** method of the Azure service management API.

```
# Python program for launching a Azure VM instance (excerpt)

from azure import *
sms = ServiceManagementService(subscription_id,
certificate_path) name = ‘<enter>'
location = 'West US'

# Name of an os image as returned by list_os_images
image_name = '<enter>'

# Destination storage account container/blob where the VM disk will be
created
media_link = <enter>'

# Linux VM configuration
linux_config = LinuxConfigurationSet('bahga', 'arshdeepbahga', 'Arsh~2483',
True)

os_hd = OSVirtualHardDisk(image_name, media_link)

#Create instance
sms.create_virtual_machine_deployment(service_name=na
   me, deployment_name=name,
   deployment_slot='production',  label=name,
   role_name=name, system_config=linux_config,
   os_virtual_hard_disk=os_hd, role_size='Small')
```

Azure Storage – Python Example

- Azure Blobs service allows you to store large amounts of **unstructured text or binary data** such as video, audio and images.
- This shows an example of using the Blob service for storing a file.
- **Blobs are organized in containers**. The **create_container** method is used to create a new container.
- After creating a container the blob is uploaded using the **put_blob** method.
- Blobs can be listed using the **list_blobs** method.
- To download a blob, the **get_blob** method is used.

```
# Python example of using Azure Blob Service (excerpt)

from azure.storage import *
blob_service = BlobService(account_name=‘enter', account_key=‘<enter>’)

#Create Container
blob_service.create_container('mycontainer')

#Upload Blob
filename='images.txt'
myblob = open(filename, 'r').read()
blob_service.put_blob('mycontainer', filename,
myblob, x_ms_blob_type='BlockBlob')

#List Blobs
blobs =
blob_service.list_blobs('mycontainer') for
blob in blobs:
   print(blob.name)
   print(blob.url)

#Download Blob
output_filename='output.txt'
blob = blob_service.get_blob('mycontainer',
'myblob') with open(output_filename, 'w') as f:
   f.write(blob)
```

Python for MapReduce

- The example shows **inverted index mapper program.**
- The **map function** reads the data from the standard input (stdin) and splits the tab-limited data into **document-ID** and contents of the document.
- The map function emits key-value pairs where key is each word in the document and value is the document-ID.

#Inverted Index Mapper in Python

```
#!/usr/bin/env python
import sys
for line in sys.stdin:
    doc_id, content = line.split('')
    words = content.split()
    for word in words:
        print '%s%s' % (word, doc_id)
```

Python for MapReduce

- The example shows inverted **index reducer program.**
- The key-value pairs emitted by the map phase are shuffled to the reducers and grouped by the key.
- The reducer reads the key-value pairs grouped by the same key from the standard input (stdin) and creates a list of document-IDs in which the word occurs.
- The output of reducer contains key value pairs where key is a unique word and value is the list of document-IDs in which the word occurs.

```
#Inverted Index Reducer in Python

#!/usr/bin/env
python import sys
current_word =
None
current_docids = []
word = None

for line in sys.stdin:
	# remove leading and trailing
	whitespace line = line.strip()
	# parse the input we got from
	mapper.py word, doc_id =
	line.split('')
	if current_word == word:
		current_docids.append(do
		c_id)
	else:
		if current_word:
			print '%s%s' % (current_word,
			current_docids) current_docids = []
			current_docids.append(doc_id)
			current_word = word
```

Python Packages of Interest

- **JSON**
  - JavaScript Object Notation (JSON) is an easy to read and write data-interchange format. JSON is used as an alternative to XML and is is easy for machines to parse and generate. JSON is built on two structures - a collection of name-value pairs (e.g. a Python dictionary) and ordered lists of values (e.g.. a Python list).
- **XML**
  - XML (Extensible Markup Language) is a data format for structured document interchange. The Python minidom library provides a minimal implementation of the Document Object Model interface and has an API similar to that in other languages.
- **HTTPLib & URLLib**
  - HTTPLib2 and URLLib2 are Python libraries used in network/internet programming
- **SMTPLib**
  - Simple Mail Transfer Protocol (SMTP) is a protocol which handles sending email and routing e-mail between mail servers. The Python smtplib module provides an SMTP client session object that can be used to send email.
- **NumPy**
  - NumPy is a package for scientific computing in Python. NumPy provides support for large multi-dimensional arrays and matrices
- **Scikit-learn**
  - Scikit-learn is an open source machine learning library for Python that provides implementations of various machine learning algorithms for classification, clustering, regression and dimension reduction problems.

Python Web Application Framework - Django

- Django is an **open source web application framework** for **developing web applications** in Python.
- A web application framework in general is a **collection of solutions, packages and best practices** that allows development of web applications and dynamic websites.
- Django is based on the **Model-Template-View architecture** and provides a separation of the data model from the business rules and the user interface.
- Django provides a **unified API** to a database backend.
- Thus web applications built with Django can work with different databases **without requiring any code changes.**
- With this fiexibility in web application design combined with the powerful capabilities of the Python language and the Python ecosystem, **Django is best suited for cloud applications.**
- Django consists of an **object-relational mapper, a web templating system and a regular-expression-based URL dispatcher.**

Django Architecture

- Django is **Model-Template-View (MTV)** framework.

- **Model**
  - The model acts as a definition of some stored data and handles the interactions with the database. In a web application, the data can be stored in a relational database, non-relational database, an XML file, etc. A Django model is a Python class that outlines the variables and methods for a particular type of data.
- **Template**
  - In a typical Django web application, the template is simply an HTML page with a few extra placeholders. Django's template language can be used to create various forms of text files (XML, email, CSS, Javascript, CSV, etc.)
- **View**
  - The view ties the model to the template. The view is where you write the code that actually generates the web pages. View determines what data is to be displayed, retrieves the data from the database and passes the data to the template.

Django Setup on Amazon EC2

Cloud Application Development in Python

Outline

- Design Approaches
  - Design methodology for IaaS service model
  - Design methodology for PaaS service model
- Cloud application case studies including:
  - Image Processing App
  - Document Storage App
  - MapReduce App
  - Social Media Analytics App

Design methodology for IaaS service model

Component Design

- Indentify the **building blocks** of the application and **to be performed by each block**
- Group the building blocks based on the functions performed and type of cloud resources required and identify the application components based on the groupings
- Identify the **inputs and outputs** of each component
- **List the interfaces** that each component will expose
- Evaluate the implementation alternatives for each component (design patterns such as MVC, etc.)

Architecture Design

- Define the interactions between the application components
- Guidelines for loosely coupled and stateless designs - use messaging queues (for asynchronous communication), functional interfaces (such as REST for loose coupling) and external status database (for stateless design)

Deployment Design

- Map the application components to specific cloud resources (such as web servers, application servers, database servers, etc.)

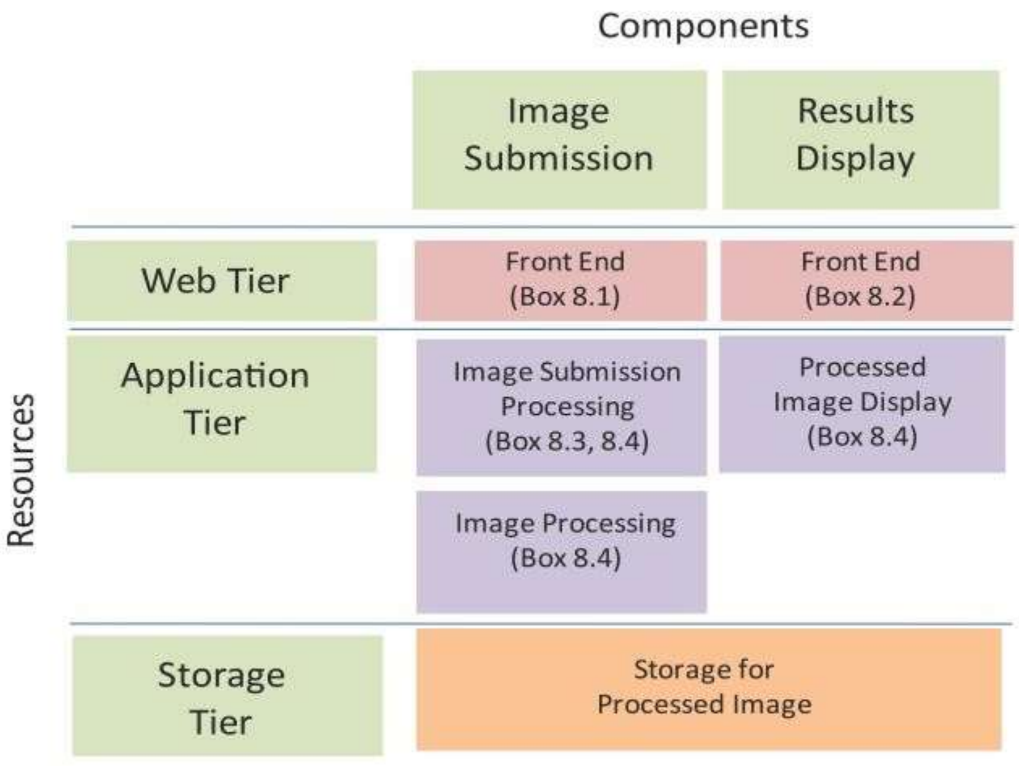
Design methodology for PaaS service model

- For applications that use the Platform-as-a-service (PaaS) cloud service model, the architecture and deployment design steps are not required since the platform takes care of the architecture and deployment.
- **Component Design**
  - In the component design step, the developers have to take into consideration the platform specific features.
- **Platform Specific Software**
  - Different PaaS offerings such as Google App Engine, Windows Azure Web Sites, etc., provide platform specific software development kits (SDKs) for developing cloud applications.
- **Sandbox Environments**
  - Applications designed for specific PaaS offerings run in sandbox environments and are allowed to perform only those actions that do not interfere with the performance of other applications.
- **Deployment & Scaling**
  - The deployment and scaling is handled by the platform while the developers focus on the application development using the platform-specific SDKs.
- **Portability**
  - Portability is a major constraint for PaaS based applications as it is difficult to move the

Image Processing App – Component Design

- **Functionality:**
  - A cloud-based Image Processing application.
  - This application provides online image filtering capability.
  - Users can upload image files and choose the filters to apply.
  - The selected filters are applied to the image and the processed image can then be downloaded.

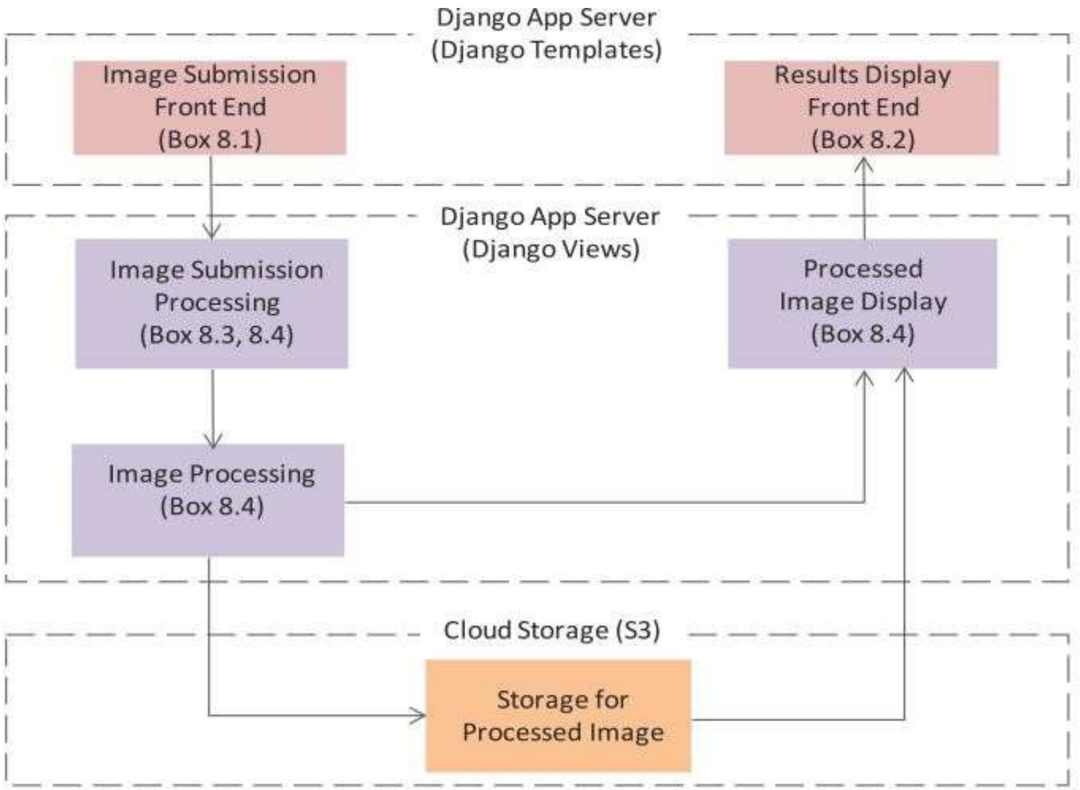
- **Component Design**
  - **Web Tier:** The web tier for the image processing app has front ends for image submission and displaying processed images.
  - **Application Tier:** The application tier has components for processing the image submission requests, processing the submitted image and processing requests for displaying the results.
  - **Storage Tier:** The storage tier comprises of the storage for processed images.



Component design for Image Processing App

Image Processing App – Architecture Design

- Architecture design step which defines the interactions between the application components.
- This application uses the Django framework, therefore, the web tier components map to the Django templates and the application tier components map to the Django views.
- A cloud storage is used for the storage tier. For each component, the corresponding code box numbers are mentioned.

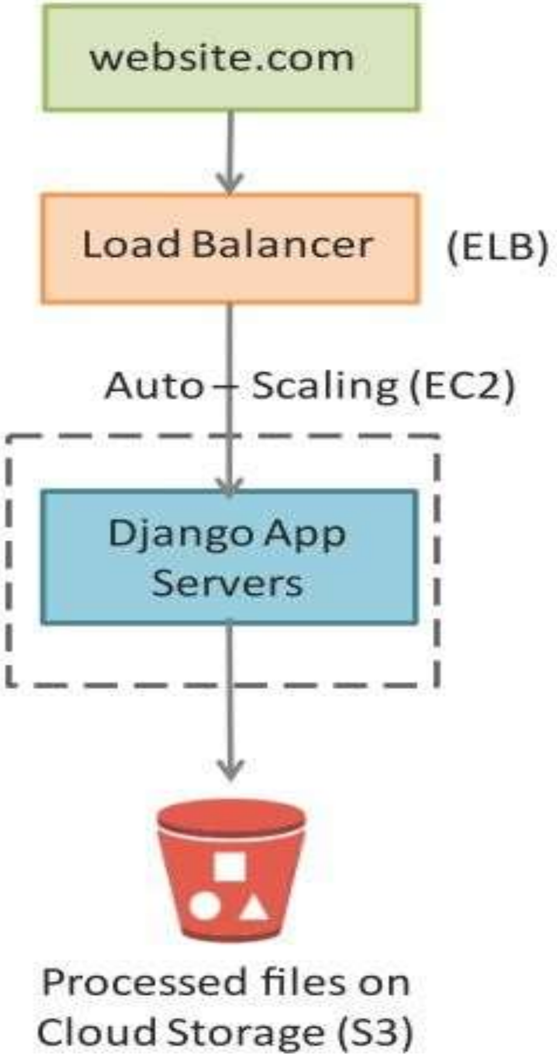


Architecture design for Image Processing App

Image Processing App – Deployment Design

- Deployment for the app is a multi-tier architecture comprising of load balancer, application servers and a cloud storage for processed images.

- For each resource in the deployment the corresponding Amazon Web Services (AWS) cloud service is mentioned.

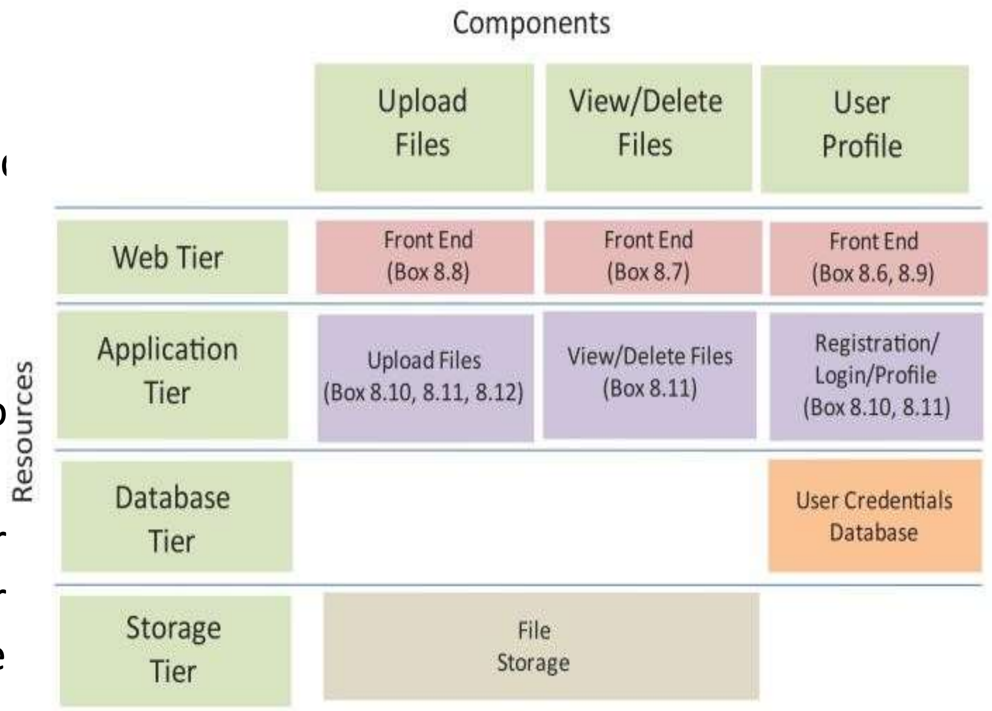


Deployment design for Image Processing App

Cloud Drive App – Component Design

- **Functionality:**
  - A cloud-based document storage (Cloud Drive) application.
  - This application allows users to store documents on a cloud- based storage.

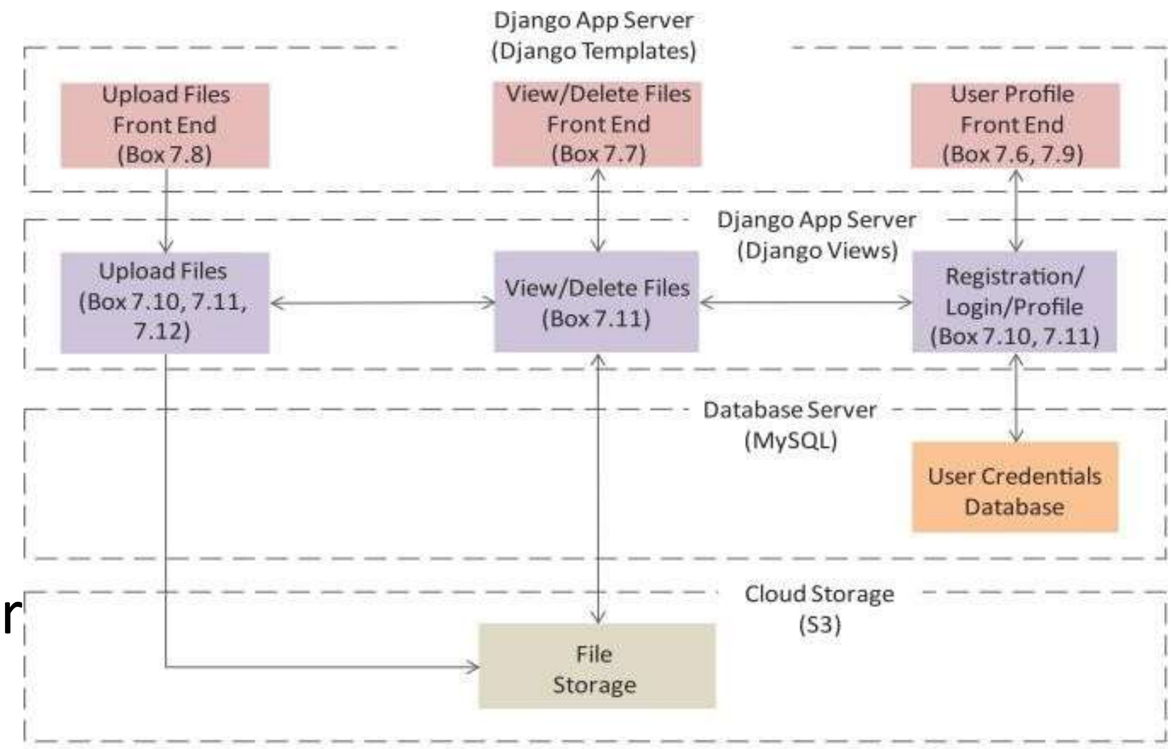
- **Component Design**
  - **Web Tier:** The web tier for the Cloud Drive app has front ends for uploading files, viewing/deleting files and user profile.
  - **Application Tier:** The application tier has components for processing requests for uploading files, processing requests for viewing/deleting files and the component that handles the registration, profile and login functions.
  - **Database Tier:** The database tier comprises of a user credentials database.
  - **Storage Tier:** The storage tier comprises of the storage for fi les.



Component design for Cloud Drive App

Cloud Drive App – Architecture Design

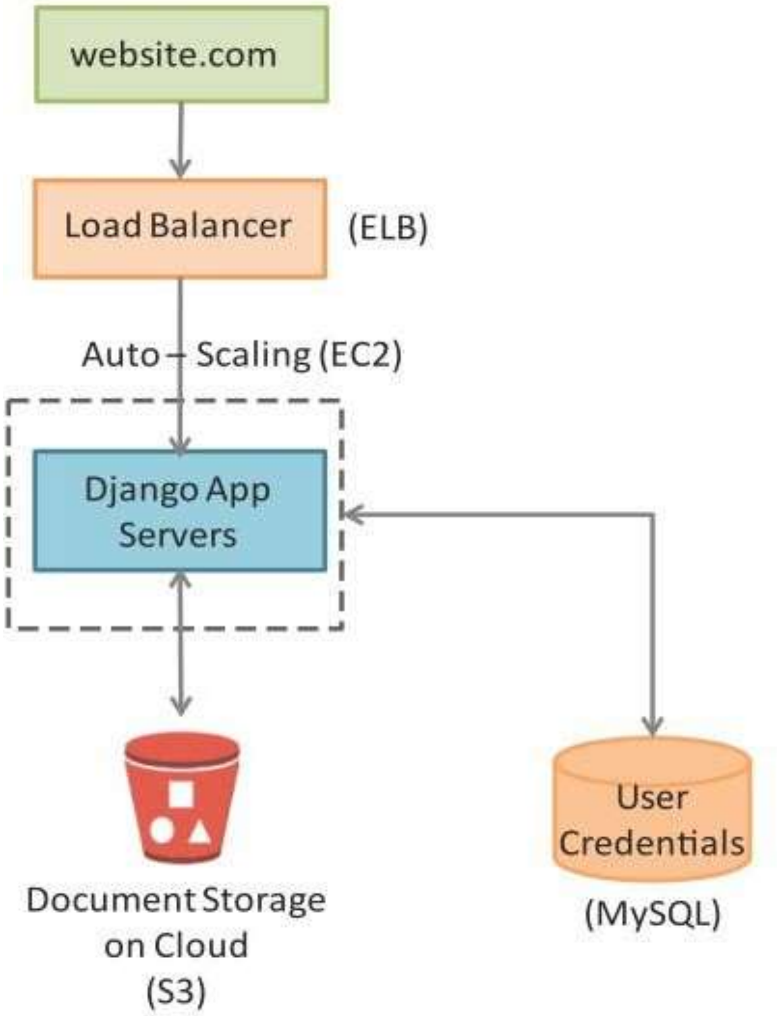
- Architecture design step which defines the interactions between the application components.
- This application uses the Django framework, therefore, the web tier components map to the Django templates and the application tier components map to the Django views.
- A MySQL database is used for the database tier and a cloud storage is used for the storage tier.
- For each component, the corresponding code box numbers are mentioned.



Architecture design for Cloud Drive App

Cloud Drive App – Deployment Design

- Deployment for the app is a multi-tier architecture comprising of load balancer, application servers, cloud storage for storing documents and a database server for storing user credentials.
- For each resource in the reference architecture the corresponding Amazon Web Services (AWS) cloud service is mentioned.



Deployment design for Cloud Drive App

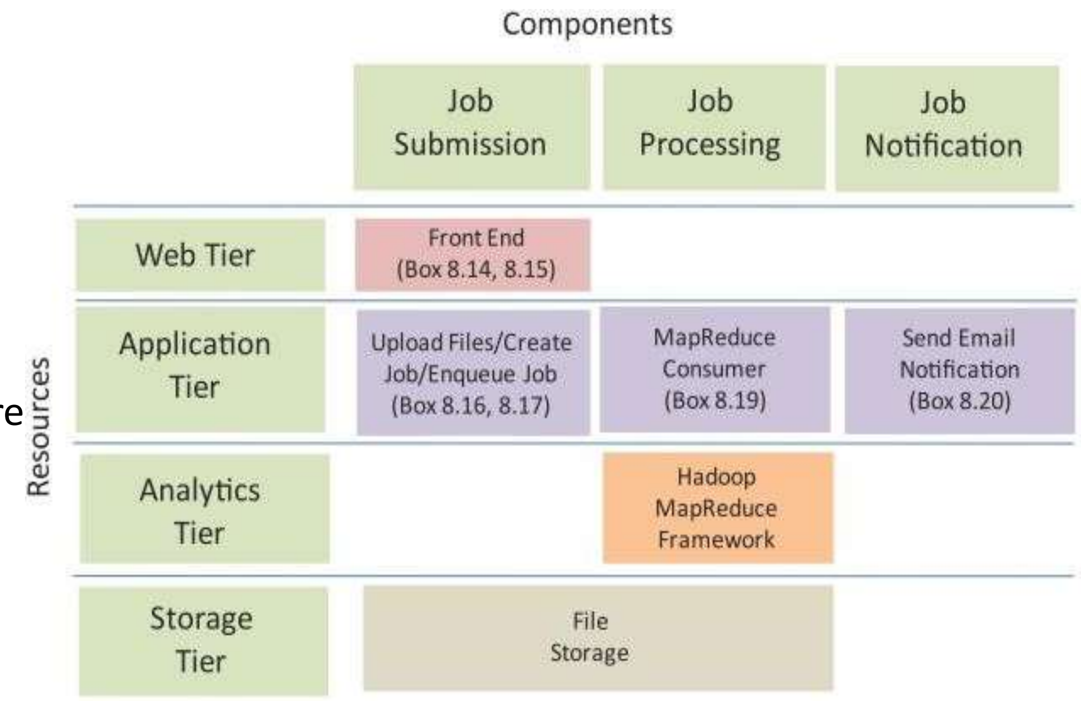
MapReduce App – Component Design

Functionality:

- This application allows users to submit MapReduce jobs for data analysis.
- This application is based on the Amazon Elastic MapReduce (EMR) service.
- Users can upload data files to analyze and choose/upload the Map and Reduce programs.
- The selected Map and Reduce programs along with the input data are submitted to a queue for processing.

Component Design

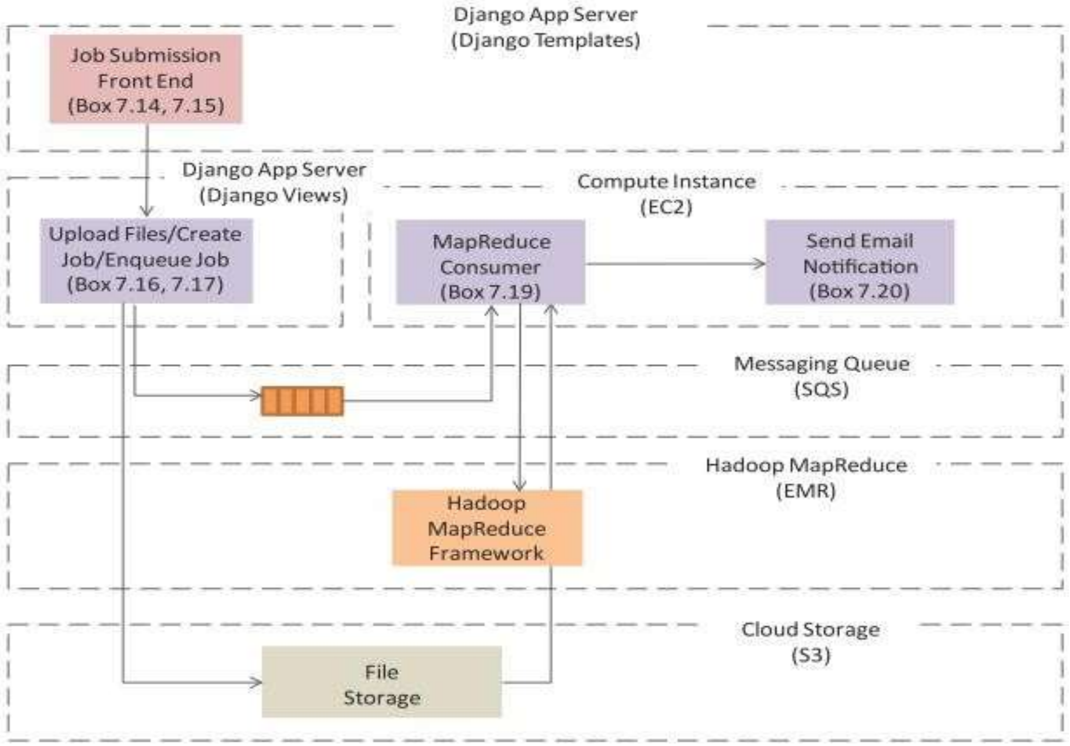
- **Web Tier:** The web tier for the MapReduce app has a front end for MapReduce job submission.
- **Application Tier:** The application tier has components for processing requests for uploading files, creating MapReduce jobs and enqueuing jobs, MapReduce consumer and the component that sends email notifications.
- **Analytics Tier:** The Hadoop framework is used for the analytics tier and a cloud storage is used for the storage tier.
- **Storage Tier:** The storage tier comprises of the storage for files.



Component design for MapReduce App

MapReduce App – Architecture Design

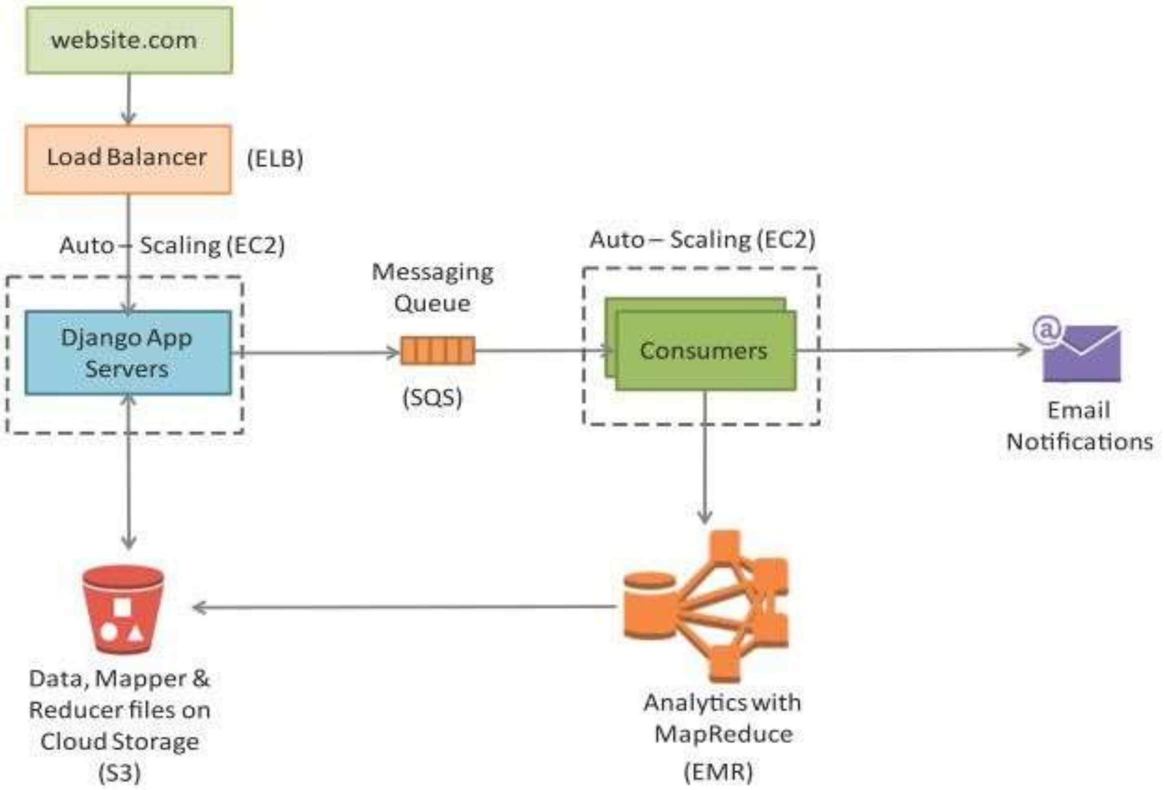
- Architecture design step which defines the interactions between the application components.
- This application uses the Django framework, therefore, the web tier components map to the Django templates and the application tier components map to the Django views.
- For each component, the corresponding code box numbers are mentioned.
- To make the application scalable the job submission and job processing components are separated.
- The MapReduce job requests are submitted to a queue.
- A consumer component that runs on a separate instance retrieves the MapReduce job requests from the queue and creates the MapReduce jobs and submits them to the Amazon EMR service.
- The user receives an email notification with the download link for the results when the job is complete.



Architecture design for MapReduce App

MapReduce App – Deployment Design

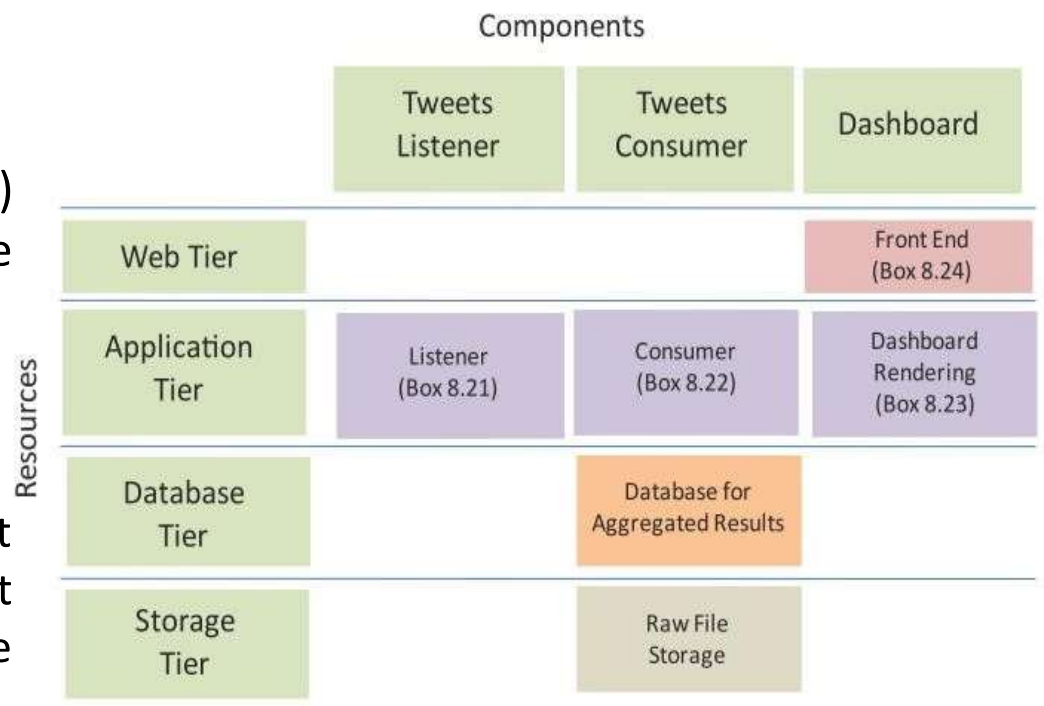
- Deployment for the app is a multi-tier architecture comprising of load balancer, application servers and a cloud storage for storing MapReduce programs, input data and MapReduce output.
- For each resource in the deployment the corresponding Amazon Web Services (AWS) cloud service is mentioned.



Deployment design for MapReduce App

Social Media Analytics App – Component Design

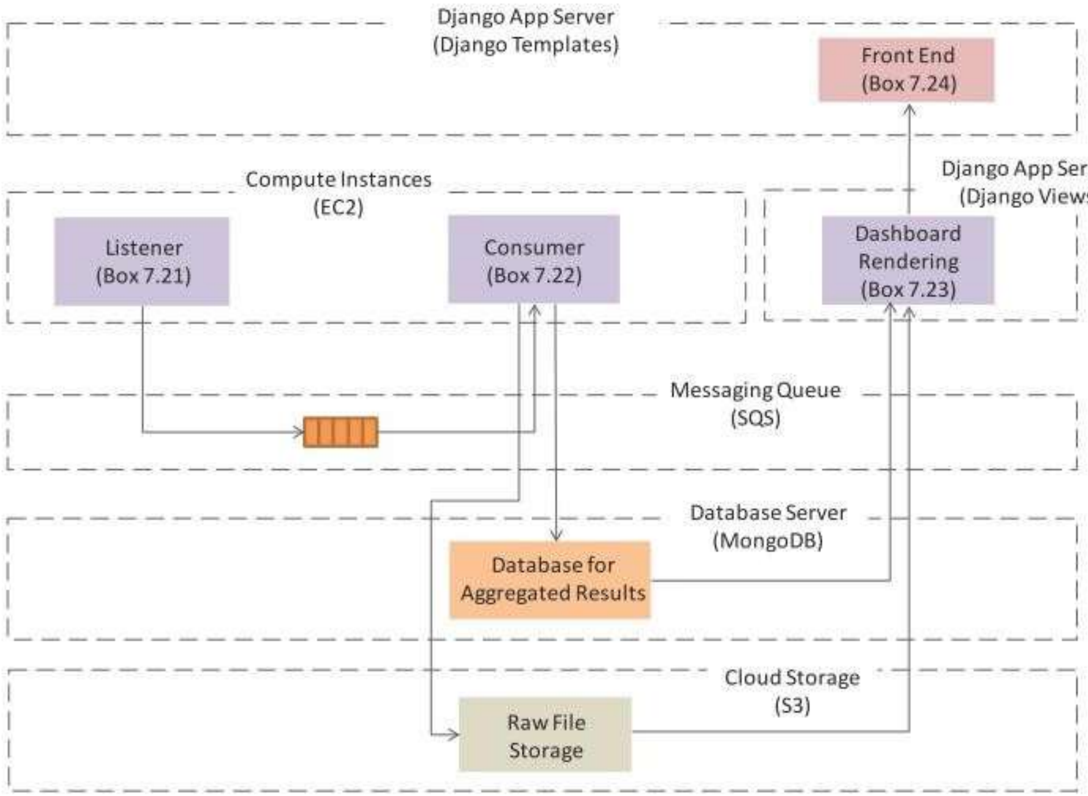
- **Functionality:**
  - A cloud-based Social Media Analytics application.
  - This application collects the social media feeds (Twitter tweets) on a specified keyword in real time and analyzes the sentiments of the tweets and provides aggregate results.
- **Component Design**
  - **Web Tier:** The web tier has a front end for displaying results.
  - **Application Tier:** The application tier has a listener component that collects social media feeds, a consumer component that analyzes tweets and a component for rendering the results in the dashboard.
  - **Database Tier:** A MongoDB database is used for the database tier and a cloud storage is used for the storage tier.
  - **Storage Tier:** The storage tier comprises of the storage for files.



Component design for Social Media Analytics App

Social Media Analytics App – Architecture Design

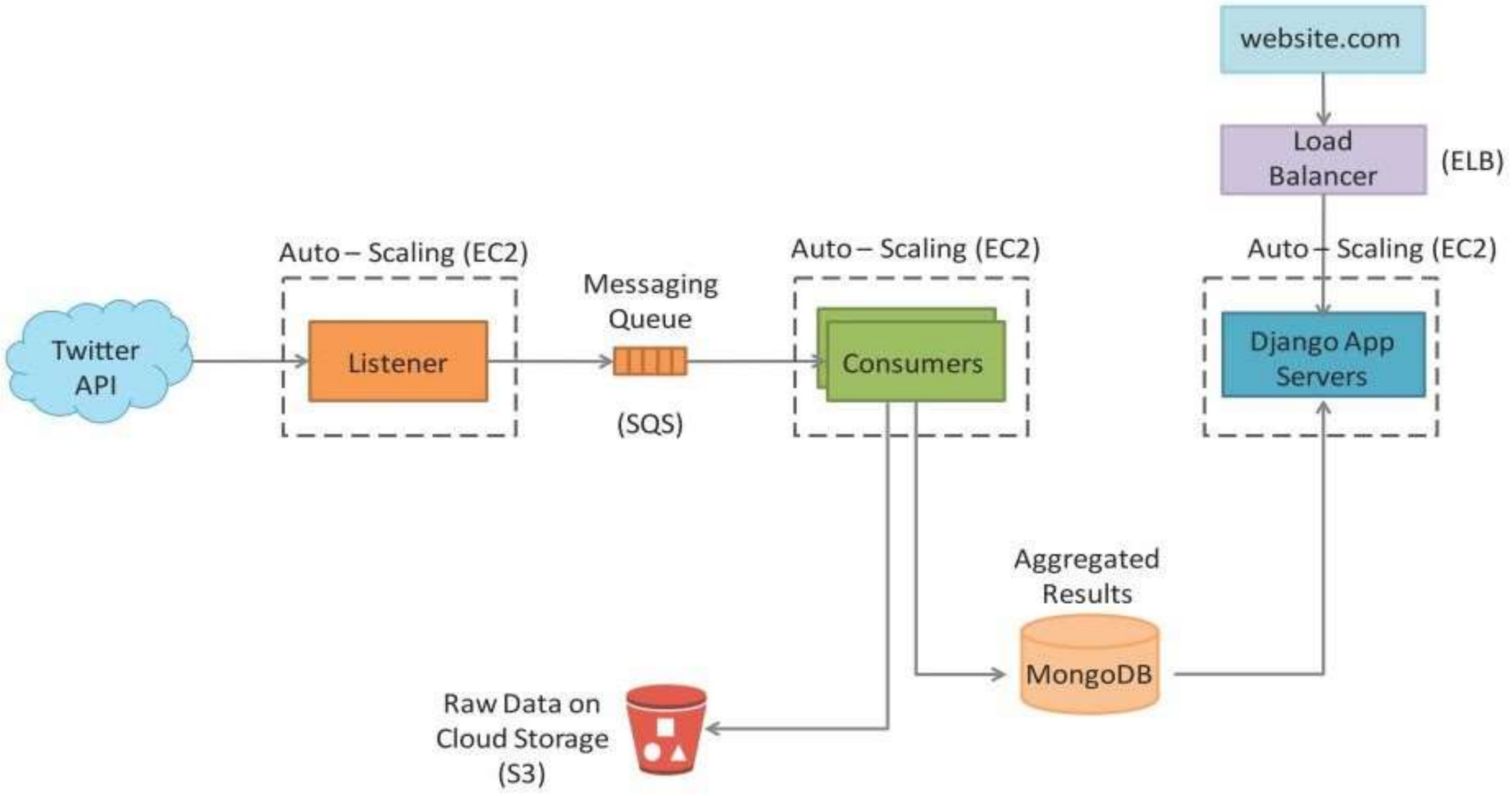
- Architecture design step which defines the interactions between the application components.
- To make the application scalable the feeds collection component (Listener) and feeds processing component (Consumer) are separated.
- The Listener component uses the Twitter API to get feeds on a specific keyword (or a list of keywords) and enqueues the feeds to a queue.
- The Consumer component (that runs on a separate instance) retrieves the feeds from the queue and analyzes the feeds and stores the aggregated results in a separate database.
- The aggregate results are displayed to the users from a Django application.



Architecture design for Social Media Analytics App

Social Media Analytics App – Deployment Design

- Deployment for the app is a multi-tier architecture comprising of load balancer, application servers, listener and consumer instances, a cloud storage for storing raw data and a database server for storing aggregated results.
- For each resource in the deployment the corresponding Amazon Web Services (AWS) cloud service is mentioned.



Deployment design for Social Media Analytics App

Social Media Analytics App – Dashboard