USN

Internal Assessment Test 2 – May 2025

Sub:	**Cloud Computing & Security**			Sub Code:	**BIS613D**	Branch:	**AIML & CSE-AIML**	
Date:	**/05/25**	Duration:	**90 minutes**	Max Marks:	**50**	Sem/Sec:	**VI**	**OBE**

		Answer any FIVE FULL Questions	**MARKS**	**CO**	**RBT**
1	a	With a neat diagram, explain the data flow in running a MapReduce job at various task trackers using Hadoop Library.	[10]	4	L2
2	a	Explain the purpose and key features of Amazon Simple Storage Service (Amazon S3)?	[10]	5	L2
3	a	Apply your knowledge of cloud platforms to describe how you would develop and deploy a web application using Google App Engine.	[10]	5	L3
4	a	Describe the Manjrasoft Aneka Cloud platform **or** OpenStack Nova System architecture with neat diagram ***(Any one)***	[10]	5	L3
5	a	Discuss how virtual machines are secured?	[05]	4	L2
	b	Explain reputation system design options.	[05]	4	L2
6	a	Apply the concept of inter-cloud resource management to solve a resource allocation problem involving multiple cloud service providers	[10]	3	L3

Q.1 a With a neat diagram, explain the data flow in running a MapReduce job at various task trackers using Hadoop Library.

Data Flow in Running a MapReduce Job Using Hadoop Library
Components involved:

- **Job Tracker:** Master node that coordinates the MapReduce job.
- **Task Trackers:** Slave nodes that run individual Map and Reduce tasks.
- **HDFS (Hadoop Distributed File System):** Stores input and output data.
- **Hadoop Library:** Provides the APIs and framework to run MapReduce jobs.

Step-by-step data flow:

1. **Job Submission:**
   - The client submits a MapReduce job to the **Job Tracker**.
   - The job includes user-written Map and Reduce functions, along with input/output paths.
2. **Job Initialization:**
   - The Job Tracker splits the input data (from HDFS) into logical splits.
   - It schedules Map tasks on Task Trackers, ideally on nodes where data blocks reside (data locality).
3. **Map Task Execution:**
   - Each Task Tracker runs Map tasks on the input split assigned.
   - The **Map function** processes input key-value pairs and produces intermediate key-value pairs.
   - Intermediate data is stored locally on the Task Tracker node.
4. **Shuffle and Sort:**

    - After Map tasks complete, Task Trackers start transferring intermediate data (shuffle phase) to the Task Trackers running Reduce tasks.
    - The intermediate key-value pairs are sorted by key during this phase.
5. **Reduce Task Execution:**
    - Reduce Task Trackers merge and process the intermediate data.
    - The **Reduce function** aggregates or processes intermediate key-value pairs to produce final output key-value pairs.
    - Output is written back to HDFS.
6. **Job Completion:**
    - The Job Tracker monitors task progress and retries failed tasks.
    - After all Reduce tasks complete, the job is marked done, and the client is notified.

Neat Diagram illustrating data flow:

```
Client
  |
  | Submit Job (MapReduce code + Input/Output paths)
  v
Job Tracker (Master)
  |
  | Assign Map Tasks on Task Trackers (Data locality)
  v
+-------------------+     +-------------------+     +-------------------+
|  Task Tracker 1   |     |  Task Tracker 2   |     |  Task Tracker N   |
|  (Map Task)       |     |  (Map Task)       |     |                   |
| Input Split 1     |     | Input Split 2     |     |                   |
| Map Function      |     | Map Function      |     |                   |
| Intermediate Data |     | Intermediate Data |     |                   |
+-------------------+     +-------------------+     +-------------------+
        |                         |                         |
        | Shuffle (Transfer intermediate data)              |
        -----------------------------------------------------
                                  |
                        +-------------------+
                        |  Task Tracker M   |
                        |  (Reduce Task)    |
                        | Shuffle + Sort    |
                        | Reduce Function   |
                        | Write output to   |
                        | HDFS              |
                        +-------------------+
                                  |
                             Job Tracker
                                  |
                      Job Completion Notification
                                  |
                                Client
```

Summary:

- The **Job Tracker** coordinates task scheduling and monitors progress.

- **Task Trackers** execute Map and Reduce tasks using the Hadoop Library.
- Data moves from HDFS to Map tasks, intermediate data is shuffled between Map and Reduce tasks, and final output is written back to HDFS.
- Hadoop Library abstracts the complexity, providing APIs to implement Map and Reduce functions and manage data flow.

Q.2 a Explain the purpose and key features of Amazon Simple Storage Service (Amazon S3)

Purpose of Amazon S3
Amazon S3 (Simple Storage Service) is a highly scalable, durable, and secure object storage service provided by AWS (Amazon Web Services). Its main purpose is to **store and retrieve any amount of data at any time, from anywhere on the web**.
It is designed for:

- **Backup and restore**
- **Archiving data**
- **Hosting static websites**
- **Big data analytics**
- **Content storage and distribution**
- **Disaster recovery**

It provides developers and enterprises with a reliable and cost-effective way to store vast amounts of data without worrying about infrastructure management.

Key Features of Amazon S3

1. **Scalability**
    - Automatically scales storage up or down based on demand.
    - Handles from a few bytes to exabytes of data.
2. **Durability and Availability**
    - Designed for **99.999999999% (11 9's) durability** by redundantly storing data across multiple geographically separated Availability Zones (AZs).
    - High availability ensures your data is accessible whenever needed.
3. **Security**
    - Supports encryption of data at rest and in transit (e.g., SSL/TLS, server-side encryption).
    - Provides fine-grained access control using IAM policies, bucket policies, and Access Control Lists (ACLs).
    - Integration with AWS Identity and Access Management (IAM) for secure authentication and authorization.
4. **Data Management Features**
    - Lifecycle policies to automate transition of objects between storage classes or delete them after a specified time.
    - Versioning to keep multiple versions of an object for backup and recovery.
    - Cross-Region Replication (CRR) to automatically replicate data to another AWS region for disaster recovery.
5. **Storage Classes**
    - Offers multiple storage classes optimized for different use cases and cost-efficiency, e.g.:
        - **S3 Standard:** Frequent access, low latency.
        - **S3 Intelligent-Tiering:** Automatically moves data between two tiers based on access patterns.

        - **S3 Standard-IA (Infrequent Access):** For data accessed less frequently but requires rapid access.
        - **S3 Glacier & Glacier Deep Archive:** For long-term archival with retrieval times ranging from minutes to hours.

6. **Performance**
    - Supports parallel uploads and downloads.
    - Provides low latency and high throughput for big data workloads.
7. **Easy Integration and Accessibility**
    - Accessible via REST API, AWS SDKs, AWS CLI, and Management Console.
    - Supports integration with other AWS services like Lambda, CloudFront, Athena, and more.
8. **Cost-Effective**
    - Pay-as-you-go pricing model.
    - No upfront costs, only pay for storage used and requests made.
    - Lifecycle policies help reduce cost by moving data to cheaper storage classes.

Summary Table

Feature	**Description**
Scalability	Automatically scales to meet storage demands
Durability	11 9's durability by multi-AZ replication
Security	Encryption, IAM integration, fine-grained access control
Storage Classes	Multiple classes for different access patterns & cost
Lifecycle Management	Automated transition and deletion of objects
Versioning	Maintain multiple versions of an object
Cross-Region Replication	Automatic data replication across regions
Performance	High throughput, low latency
Integration	APIs, SDKs, AWS service compatibility
Cost	Pay-as-you-go with cost-optimization features

Q.3 a Apply your knowledge of cloud platforms to describe how you would develop and deploy a web application using Google App Engine.

Developing and Deploying a Web Application Using Google App Engine
What is Google App Engine?
Google App Engine is a fully managed Platform-as-a-Service (PaaS) by Google Cloud that allows developers to build and deploy scalable web applications and services without managing the underlying infrastructure.

Step 1: Set Up Google Cloud Environment

- **Create a Google Cloud Project:**
  Go to the Google Cloud Console and create a new project. This project will host your application and associated resources.
- **Enable Billing:**
  Link your project to a billing account because GAE requires billing for deployment, though there is a free tier for limited usage.
- **Enable App Engine API:**
  Activate the App Engine API from the Cloud Console to use GAE services.

Step 2: Develop Your Web Application

- **Choose your language/runtime:**
  Google App Engine supports several runtimes: Python, Java, Node.js, Go, PHP, Ruby, etc. Choose one based on your application needs.
- **Write your application code:**
  Develop your web app using your preferred framework (e.g., Flask or Django for Python, Express for Node.js).
- **Create the app.yaml configuration file:**
  This YAML file tells GAE how to deploy and run your application — runtime environment, instance class, scaling type, handlers, etc.

Step 3: Test Locally

- Use the Google Cloud SDK to run your app locally for testing:

```
dev_appserver.py app.yaml
```

- Verify your app runs as expected before deploying.

Step 4: Deploy Your Application

- Install and configure **Google Cloud SDK** if not already installed.
- Authenticate with your Google Cloud account:

```
gcloud auth login
```

- Set the project:

```
gcloud config set project [YOUR_PROJECT_ID]
```

- Deploy the app using:

```
gcloud app deploy
```

- This uploads your code, dependencies, and config to Google Cloud and starts your app on App Engine.

Step 5: Access and Monitor

- After deployment, access your app at:

https://[YOUR_PROJECT_ID].appspot.com

- Use the Google Cloud Console to monitor:
  - Traffic and performance
  - Logs via Cloud Logging
  - Scaling behavior and resource usage

Step 6: Scaling and Managing

- Google App Engine automatically handles **scaling** depending on traffic:
  - **Automatic Scaling** (default) — instances spin up/down based on requests.
  - **Manual or Basic Scaling** can also be configured in app.yaml.
- Use Cloud Console or CLI to update your app or rollback to previous versions.

Summary Flow:

1. Create GCP project →
2. Write web app code + app.yaml →
3. Test locally →

4. Deploy with gcloud app deploy →
5. Access app on App Engine URL →
6. Monitor & manage scaling/logs.

Q.4 a Describe the Manjrasoft Aneka Cloud platform or OpenStack Nova System architecture with neat diagram (Any one)

OpenStack Nova System Architecture
Purpose:
Nova provides the compute resources—virtual servers/instances—that users request via the OpenStack API or dashboard. It manages the lifecycle of compute instances in a cloud environment.

Key Components of Nova:

1. **Nova API Server**
    - Accepts and processes REST API requests from users.
    - Validates and authenticates requests before forwarding them to other components.
2. **Scheduler**
    - Responsible for selecting the best compute node to run a new VM based on resource availability and policies.
3. **Compute Nodes**
    - Physical servers running hypervisors (e.g., KVM, Xen, VMware) that actually host VM instances.
    - Each compute node runs an **nova-compute** service responsible for managing VM lifecycle on that node.
4. **Conductor**
    - Mediates interactions between Nova components and the database.
    - Helps avoid direct database access by other services.
5. **Message Queue**
    - Nova uses a message queue (e.g., RabbitMQ) to communicate asynchronously between components.
6. **Database**
    - Stores state and metadata about instances, flavors, projects, quotas, etc.
7. **Network Service**
    - Integrates with OpenStack Neutron to provide networking to instances.

Workflow Summary:

- User sends a request to create an instance via Nova API.
- Nova API authenticates and forwards the request to the Scheduler.
- Scheduler selects an appropriate Compute Node.
- The Compute Node's nova-compute service communicates with the hypervisor to create the VM.
- Nova services use message queues for inter-service communication.
- The database stores and updates instance state information.
- Networking is set up via Neutron to attach the VM to networks.

Neat Diagram of OpenStack Nova Architecture:

```
          +---------------------+
          |    User / Client    |
            +----------+----------+
                     |
             REST API Calls (Nova API)
                     |
            +----------v----------+
            |    Nova API Server  |
            +----------+----------+
                     |
            Request forwarded to Scheduler
                     |
            +----------v----------+
            |     Scheduler     |
            +----------+----------+
                     |
        +----------------+----------------+
        |                          |
+---------v---------+          +---------v---------+
|  Compute Node 1  |           |  Compute Node N   |
| nova-compute svc  |          | nova-compute svc  |
| Hypervisor (KVM)  |          | Hypervisor (KVM)  |
+---------+---------+          +---------+---------+
        |                          |
     Manages VM instances          Manages VM instances
        |                          |
     +---+---+                   +---+---+
     | VMs    |                  | VMs   |
     +--------+                  +-------+
```

Other Supporting Components:

```
+---------------------+    +--------------------+    +--------------------+
|    Message Queue    |<--->|     Conductor     |<--->|      Database      |
+---------------------+    +--------------------+    +--------------------+
```

Network services (Neutron) provide VM networking and are integrated with Nova.

Summary:

- Nova is the compute service for managing VM lifecycle.
- API server handles requests, Scheduler assigns nodes.
- Compute nodes run hypervisors hosting VMs.
- Communication is via message queues.
- Database stores metadata.
- Works closely with networking (Neutron) and storage components.

Q.5 a Discuss how virtual machines are secured?

How Virtual Machines Are Secured

Virtual Machines run on shared physical hardware but need isolation and protection to ensure security. Here’s how VM security is typically ensured:

---

1. Isolation

- **Hypervisor Isolation:**
  The hypervisor (also called Virtual Machine Monitor - VMM) enforces strict separation between VMs running on the same host, preventing direct access to each other’s memory or CPU resources.
- **Network Isolation:**
  Use of virtual LANs (VLANs), virtual switches, or software-defined networking (SDN) to segment VM network traffic, reducing risk of network-based attacks.

---

2. Access Control

- **Authentication & Authorization:**
  Only authorized users can access VM consoles, management interfaces, or APIs. Integration with centralized identity management (e.g., LDAP, Active Directory).
- **Role-Based Access Control (RBAC):**
  Define roles and permissions so users can only perform actions appropriate to their role.

---

3. Patch Management and Hardening

- **Regular Updates:**
  Keep VM operating systems and applications up to date with security patches.
- **OS & Application Hardening:**
  Disable unnecessary services, use firewalls inside VMs, and configure security settings per best practices.

---

4. Network Security

- **Firewalls and Security Groups:**
  Configure firewalls at the VM level and use security groups (cloud-specific firewall rules) to restrict inbound/outbound traffic.
- **Intrusion Detection/Prevention Systems (IDS/IPS):**
  Deploy IDS/IPS to monitor VM network traffic for suspicious activity.

---

5. Encryption

- **Data-at-Rest Encryption:**
  Encrypt VM disk images and storage volumes to protect data in case of physical theft or unauthorized access.
- **Data-in-Transit Encryption:**
  Use protocols like TLS/SSL to encrypt data moving between VMs and other systems.

---

6. Monitoring and Logging

- **Log Monitoring:**
  Collect and analyze logs from VMs and hypervisors for unusual behavior or breaches.
- **Continuous Monitoring:**
  Use security tools to continuously check VM integrity and compliance with policies.

---

7. Backup and Recovery

- Regular backups of VM images and data ensure recovery in case of attack (e.g., ransomware) or failure.

---

8. Secure VM Lifecycle Management

- **Secure Provisioning:**
  Use trusted VM templates that are pre-hardened.
- **Decommissioning:**
  Securely delete VM data and snapshots before disposal.

---

9. Hypervisor and Host Security

- Secure the hypervisor and the host OS because if compromised, all hosted VMs can be affected.
- Apply patches, limit admin access, and monitor the host environment.

---

10. Use of Security Tools and Best Practices

- **VM Security Tools:**
  Tools like VMware NSX, Microsoft Shielded VMs, or cloud-native security services.
- **Security Frameworks:**
  Follow industry standards and guidelines (e.g., CIS Benchmarks, NIST).

Summary Table

Security Aspect	**Measures**
Isolation	Hypervisor enforced VM isolation, network segmentation
Access Control	Authentication, RBAC, strong credentials
Patch & Hardening	Regular updates, OS/app hardening
Network Security	Firewalls, security groups, IDS/IPS
Encryption	Disk and network traffic encryption
Monitoring & Logging	Log analysis, continuous security monitoring
Backup & Recovery	Regular VM backups and tested recovery plans
VM Lifecycle Management	Secure provisioning and decommissioning
Hypervisor Security	Harden host and hypervisor OS, monitor for threats

---

Q.5.b Explain reputation system design options.

Reputation System Design Options
A **reputation system** is a mechanism used in online platforms to evaluate and represent the trustworthiness, reliability, or quality of users, products, or content based on feedback, behavior, or interactions. Designing such systems involves choosing how reputation scores are calculated, updated, and used.

Key Design Options in Reputation Systems

1. **Centralized vs Decentralized Reputation Systems**

- **Centralized:**
  One central authority collects and manages all reputation data. Easier to control and moderate but can be a single point of failure or bias.
- **Decentralized:**
  Reputation data is distributed across multiple nodes or participants, improving robustness and resistance to manipulation (e.g., blockchain-based systems).

---

2. **Explicit vs Implicit Reputation**

- **Explicit:**
  Users provide direct feedback such as ratings, reviews, or thumbs-up/down (e.g., eBay feedback scores).
- **Implicit:**
  Reputation is inferred from user behavior or interactions without direct input, like frequency of transactions, response time, or social network connections.

---

3. **Global vs Local Reputation**

- **Global:**
  A single reputation score is maintained per user/entity across the entire system.
- **Local:**
  Reputation is context-specific, varying by community, category, or product (e.g., a seller might have different reputations in electronics vs clothing).

---

4. **Static vs Dynamic Reputation**

- **Static:**
  Reputation values are fixed once assigned (rarely used as it can become outdated).
- **Dynamic:**
  Reputation updates continuously or periodically based on recent behavior, giving more weight to recent activity.

---

5. **Binary vs Multilevel Reputation**

- **Binary:**
  Simple good/bad or trustworthy/untrustworthy states.
- **Multilevel:**
  Reputation represented as a score or level (e.g., a 1–5 star rating system).

---

6. **Aggregated vs Weighted Reputation**

- **Aggregated:**
  Reputation is a simple average or sum of all feedback.
- **Weighted:**
  Feedback is weighted based on factors like rater’s reputation, recency, or transaction value to reduce manipulation.

---

7. **Transparent vs Private Reputation**

- **Transparent:**
  Reputation scores and feedback are publicly visible.
- **Private:**
  Scores are only available to certain parties (e.g., the user and platform admins), to prevent targeted attacks or bias.

---

8. **Punishment and Reward Mechanisms**

- Design options for penalizing negative behavior or rewarding positive actions, which influence reputation scores accordingly.

Example Design Combinations:

Design Aspect	Example Option	Use Case/Benefit
Centralization	Centralized	Easier moderation, control
Feedback Type	Explicit	Clear user opinions

Design Aspect	Example Option	Use Case/Benefit
Reputation Scope	Local	Contextual reputation (per community)
Update Frequency	Dynamic	Reflects current user behavior
Reputation Scale	Multilevel	More nuanced reputation (1-5 stars)
Feedback Weighting	Weighted	Reduces fake feedback impact
Visibility	Transparent	Builds trust with open feedback

Q. 6 a Apply the concept of inter-cloud resource management to solve a resource allocation problem involving multiple cloud service providers

---

Inter-Cloud Resource Management: Concept Overview

Inter-cloud resource management involves coordinating and managing computing resources across **multiple cloud service providers** (CSPs) to optimize usage, improve availability, and reduce costs.

It enables:

- **Resource sharing** among clouds
- **Load balancing** across providers
- **Fault tolerance** through redundancy
- **Cost optimization** by choosing best-priced resources

Problem Scenario

Imagine a company needs to allocate computing resources (CPU, memory, storage) to run a large-scale application but wants to use multiple CSPs (e.g., AWS, Azure, Google Cloud) due to:

- Avoiding vendor lock-in
- Ensuring high availability
- Optimizing cost

Applying Inter-Cloud Resource Management to Solve Resource Allocation

Step 1: Resource Discovery

- Collect real-time data about resource availability and pricing from all CSPs via their APIs.
- Gather metrics such as current CPU loads, memory availability, network latency, and cost per resource unit.

Step 2: Define Allocation Criteria

- Define constraints and goals, e.g.:
    - Minimum resource requirements (e.g., 100 CPUs, 500 GB RAM)
    - Budget limits
    - Latency requirements (e.g., resources must be within certain geographic regions)
    - Reliability needs (e.g., redundancy across clouds)

Step 3: Resource Scheduling and Optimization

- Use a resource broker or a centralized controller that:
    - Analyzes resource availability and cost from each cloud.
    - Uses algorithms (e.g., linear programming, heuristic algorithms) to allocate workloads to the cloud(s) that best meet requirements.
    - Balances load and cost while meeting SLA constraints.

Example:

- Assign 60% workload to AWS (due to lower cost and availability), 30% to Azure (for geographic diversity), and 10% to Google Cloud (for specialized GPU instances).

Step 4: Provisioning & Deployment

- Automatically provision the selected resources on each CSP.
- Deploy application components or virtual machines accordingly.

Step 5: Monitoring & Dynamic Adjustment

- Continuously monitor resource usage, performance, and costs.
- Dynamically reallocate workloads if resource availability changes or costs fluctuate (elastic scaling).

Architecture Diagram (Conceptual)

```
+-----------------+      +-----------------+      +-----------------+
|   AWS Cloud     |      |  Azure Cloud    |      | Google Cloud    |
| (Resources, API)|      | (Resources, API)|      | (Resources, API)|
 +--------+--------+      +--------+--------+      +--------+--------+
          |                        |                        |
          +-----------+------------+------------+-----------+
                      |   Inter-Cloud Resource Broker    |
                      | (Discovery, Scheduling, Provision)|
                      +---------------+-------------------+
                                      |
                               +------+------+
                               |  Application |
                               |  Workloads   |
                               +-------------+
```

Benefits of This Approach

- **Improved resource utilization** by leveraging multiple clouds.
- **Cost savings** by choosing optimal pricing.
- **Higher availability and fault tolerance**.
- **Flexibility and scalability** by dynamically shifting loads.