Problem Statement

The effective resource management in cloud computing systems enables maximum system performance as well as resource utilization efficiency. The main difficulty in this field involves finding appropriate Virtual Machine (VM) allocation solutions to manage user-generated Cloudlets. The simulation evaluates the performance outcomes together with resource utilization and task processing duration through TimeShared (Round Robin) and SpaceShared (First-Come-First-Serve) virtual machine scheduling methods in cloud datacenters.

Objectives:

- Simulate a basic cloud environment using **CloudSim**
- Compare two VM allocation strategies:
    - TimeShared (Round Robin)
    - SpaceShared (First-Come-First-Serve)
- Measure and analyze Cloudlet execution times under each strategy
- Automate simulation execution using **GitHub Actions (CI/CD)**
- Visualize and document the results

Implementation:

CloudSim Framework Configuration:

- **Datacenter**: 1
- **Hosts**: 2 (Each with 1 CPU core, 2048MB RAM, 10000 bandwidth)
- **VMs**: 4 attempted, 2 successfully created (due to MIPS constraints)
- **Cloudlets**: 10 identical jobs (length = 40000)
- **Schedulers**:
    - `VmSchedulerTimeShared` / `CloudletSchedulerTimeShared` → Round Robin
    - `VmSchedulerSpaceShared` / `CloudletSchedulerSpaceShared` → FCFS

Step 1: Run Time-Shared and Space-Shared VM Scheduling Simulation

```
// Run Time-Shared VM scheduling simulation
System.out.println("\n===== RUNNING TIME-SHARED VM SCHEDULING SIMULATION
=====\n");
long timeSharedStartTime = System.currentTimeMillis();
List<Cloudlet> timeSharedResults = runSimulation("TimeShared", true);
long timeSharedEndTime = System.currentTimeMillis();
double timeSharedRuntime = (timeSharedEndTime - timeSharedStartTime) / 1000.0;

// Run Space-Shared VM scheduling simulation
System.out.println("\n===== RUNNING SPACE-SHARED VM SCHEDULING SIMULATION
=====\n");
long spaceSharedStartTime = System.currentTimeMillis();
List<Cloudlet> spaceSharedResults = runSimulation("SpaceShared", false);
long spaceSharedEndTime = System.currentTimeMillis();
double spaceSharedRuntime = (spaceSharedEndTime - spaceSharedStartTime) /
1000.0;
```

This part of the code runs two separate simulations—one using **TimeShared** and the other using **SpaceShared** VM scheduling. It records the start and end time of each simulation using `System.currentTimeMillis()` to calculate how long each one takes to complete. The results of both runs are stored for later comparison in terms of performance and efficiency.

Step 2: Create VMs

```
// Create VMs with appropriate scheduler
List<Vm> vmlist = new ArrayList<>();
for (int i = 0; i < 4; i++) {
    CloudletScheduler scheduler = isTimeShared ?
    new CloudletSchedulerTimeShared() : new CloudletSchedulerSpaceShared();

    Vm vm = new Vm(i, broker.getId(), 1000, 1, 512, 1000, 1000,"Xen",
scheduler);
    vmlist.add(vm);
}
```

This part of the code creates a list of Virtual Machines (VMs) and assigns a scheduling strategy to each one based on the simulation mode. It runs a loop to create 4 VMs, and for each VM, it selects either **TimeShared** or **SpaceShared** scheduling depending on the `isTimeShared` flag. Each VM is given fixed configurations such as 1000 MIPS,

512 MB RAM, and 1000 MB storage, and then added to the VM list to be submitted for simulation.

Step 3:Create Cloudlets

```
// Create Cloudlets
List<Cloudlet> cloudletList = new ArrayList<>();
for (int i = 0; i < 10; i++) {
    Cloudlet cloudlet = new Cloudlet(i, 40000, 1, 30, 30,
    new UtilizationModelFull(), new UtilizationModelFull(),
    new UtilizationModelFull());
    cloudlet.setUserId(broker.getId());
    cloudletList.add(cloudlet);
}
```

This code creates a list of 10 cloudlets, each with a length of 40,000 instructions and resource requirements of 30 units. Each cloudlet is assigned full CPU, memory, and bandwidth usage, and a user ID from the broker. The cloudlets are then added to the `cloudletList`.

Step 4: Create Datacenter

```
 List<Host> hostList = new ArrayList<>();

     for (int i = 0; i < 2; i++) {
         List<Pe> peList = new ArrayList<>();
         peList.add(new Pe(0, new PeProvisionerSimple(1000)));

         VmScheduler scheduler = isTimeShared ?
             new VmSchedulerTimeShared(peList) : new
VmSchedulerSpaceShared(peList);

         Host host = new Host(i, new RamProvisionerSimple(2048),
                 new BwProvisionerSimple(10000), 1000000, peList, scheduler);

         hostList.add(host);
     }

     DatacenterCharacteristics characteristics = new
DatacenterCharacteristics(
             "x86", "Linux", "Xen", hostList, 10.0, 3.0, 0.05, 0.1, 0.1);

     return new Datacenter(name, characteristics,
             new VmAllocationPolicySimple(hostList), new LinkedList<Storage>
(), 0);
```

This code creates a list of two hosts for a cloud data center. Each host is configured with a processor (`Pe`), a memory provisioner, a bandwidth provisioner, and a VM scheduler (either time-shared or space-shared based on the `isTimeShared` flag). The hosts are then added to `hostList`. Next, the `DatacenterCharacteristics` are defined, specifying details like architecture, operating system, hypervisor, and host list. Finally, a new `Datacenter` is created using these characteristics, a simple VM allocation policy, and no storage, and it is returned.

Step 5: Calculate Max Finish Time

```
 private static double getMaxFinishTime(List<Cloudlet> cloudlets) {
     double maxFinishTime = 0;
     for (Cloudlet cloudlet : cloudlets) {
         if (cloudlet.getFinishTime() > maxFinishTime) {
             maxFinishTime = cloudlet.getFinishTime();
         }
     }
     return maxFinishTime;
 }
```

This method calculates the maximum finish time from a list of cloudlets. It initializes `maxFinishTime` to 0, then iterates over each cloudlet in the list. For each cloudlet, it checks if its finish time is greater than the current `maxFinishTime`. If so, it updates `maxFinishTime` with the new value. Finally, the method returns the highest finish time found among all cloudlets.

Step 5: Calculate Average Execution Time

```
private static double getAverageExecutionTime(List<Cloudlet> cloudlets) {
    double totalExecutionTime = 0;
    int successfulCloudlets = 0;

    for (Cloudlet cloudlet : cloudlets) {
        if (cloudlet.getStatus() == Cloudlet.SUCCESS) {
            successfulCloudlets++;
            totalExecutionTime += cloudlet.getActualCPUTime();
        }
    }

    return successfulCloudlets > 0 ? totalExecutionTime / successfulCloudlets
: 0;
 }
```

This method calculates the average execution time of successful cloudlets from a list. It initializes `totalExecutionTime` to 0 and `successfulCloudlets` to count the cloudlets that completed successfully. It then iterates over the cloudlets, adding the actual CPU time of each successful cloudlet to `totalExecutionTime` and incrementing the `successfulCloudlets` counter. After the loop, it returns the average execution time by dividing `totalExecutionTime` by the number of successful cloudlets. If there are no successful cloudlets, it returns 0.

CI/CD Pipeline Implementation

- `.yml` workflow runs automatically on **push to `main` branch**
- Workflow steps:
    - Set up Java 11
    - Compile simulation code
    - Run the simulation
- Output is visible in **Actions tab**, ensuring:

- **Automation**
- **Consistency**
- **Error checking** on every code update

Benefits of CI/CD Implementation

- Reduced manual deployment effort
- Consistent build and test environment
- Faster release cycles
- Better code quality through automated testing
- Enhanced team collaboration

Conclusion

The project successfully simulated and compared two VM allocation strategies using CloudSim:

- **TimeShared (Round Robin)** allows better CPU sharing but causes tasks to complete at the same time
- **SpaceShared (FCFS)** executes tasks faster individually but sequentially, leading to a similar overall time

The use of **GitHub Actions** enabled full CI/CD automation, providing a modern development pipeline.

Future Scope

- Add support for dynamic resource scaling
- Visualize results using graphs or charts
- Export results to CSV or PDF
- Add a simple frontend (JavaFX / Spring Boot + HTML)