Project and Testing Report

Cloud Management System

Adham Ahmed Mekky	231001296
Adham Ahmad Abdelaal	231000313
Amr Sherif Nabil	231000045
Belal Mohammed Amer	231000272
Youssef Helmy Ibrahim	231000834

1. Project Overview

The Cloud Management System is a virtualization management tool developed in Python. It serves as a centralized dashboard that allows users to create and manage Virtual Machines (VMs) using QEMU/KVM and Containerized Applications using the Docker Engine. The system was developed on a Windows Subsystem for Linux (WSL2) environment to leverage native Linux kernel features while running on a Windows host.

2. Design Choices

To meet the project requirements, the following design decisions were made:

- **Programming Language:** Python 3 was chosen for its strong support for system automation and library ecosystem.
- **Libraries Used:**
  - `subprocess`: Used to execute complex QEMU shell commands directly from the script.
  - `docker`: The official Python SDK was used to interact with the Docker daemon programmatically, which is cleaner and more robust than running shell commands.
  - `json`: Used to parse configuration files, allowing for reproducible VM setups.
- **Architecture:** The code is modularized into three separate files (`main.py`, `vm_manager.py`, `docker_manager.py`) to separate concerns. The `main.py` acts as the controller (UI), while the other modules handle the logic.

3. Challenges Faced & Solutions

During the development process, several technical hurdles were encountered:

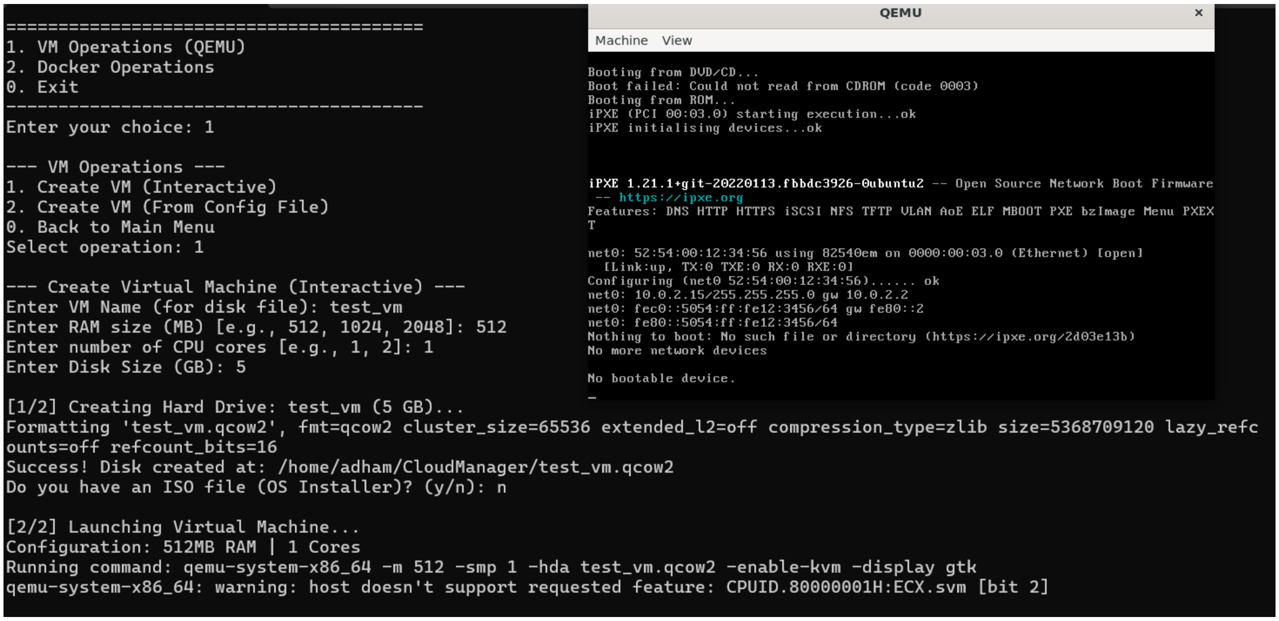
- **Challenge 1: Docker Socket Permissions**
  - *Issue:* The application initially failed with a "Permission Denied" error when trying to talk to the Docker Daemon.
  - *Solution:* We modified the user privileges by adding the current user to the `docker` group (`sudo usermod -aG docker $USER`) and restarting the session.
- **Challenge 2: KVM/QEMU Access in WSL**
  - *Issue:* QEMU failed to launch with a "Could not access KVM kernel module" error. This is a known issue in WSL where permissions reset after reboot.
  - *Solution:* We implemented a manual override by changing the permissions of the accelerator device (`sudo chmod 666 /dev/kvm`) before launching the application.
- **Challenge 3: Handling User Input**
  - *Issue:* Users could crash the program by typing text when numbers were expected (e.g., for RAM size).
  - *Solution:* We implemented `try-except` blocks to catch `ValueError` and prompt the user to try again, preventing the system from crashing.

4. Testing & Evaluation

The system was tested thoroughly to ensure all functional requirements were met. Below are the test cases and evidence of success.

Test Case 1: Virtual Machine Creation

- **Objective:** Create a VM with 512MB RAM and 1 CPU core using the interactive menu.
- **Result:** The QEMU window launched successfully. The "No bootable device" screen confirms the virtual hardware was initialized correctly.

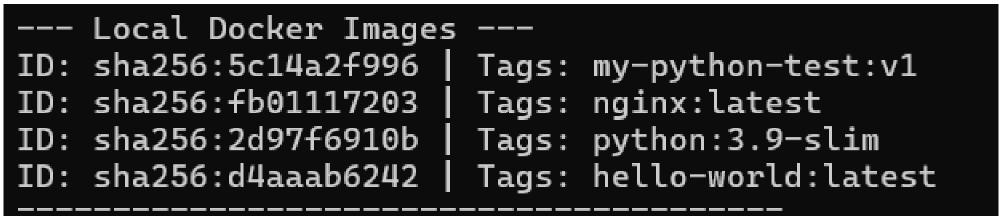


```
========================================
1. VM Operations (QEMU)
2. Docker Operations
0. Exit
----------------------------------------
Enter your choice: 1

--- VM Operations ---
1. Create VM (Interactive)
2. Create VM (From Config File)
0. Back to Main Menu
Select operation: 1

--- Create Virtual Machine (Interactive) ---
Enter VM Name (for disk file): test_vm
Enter RAM size (MB) [e.g., 512, 1024, 2048]: 512
Enter number of CPU cores [e.g., 1, 2]: 1
Enter Disk Size (GB): 5

[1/2] Creating Hard Drive: test_vm (5 GB)...
Formatting 'test_vm.qcow2', fmt=qcow2 cluster_size=65536 extended_l2=off compression_type=zlib size=5368709120 lazy_refc
ounts=off refcount_bits=16
Success! Disk created at: /home/adham/CloudManager/test_vm.qcow2
Do you have an ISO file (OS Installer)? (y/n): n

[2/2] Launching Virtual Machine...
Configuration: 512MB RAM | 1 Cores
Running command: qemu-system-x86_64 -m 512 -smp 1 -hda test_vm.qcow2 -enable-kvm -display gtk
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bit 2]
```

```
Booting from DVD/CD...
Boot failed: Could not read from CDROM (code 0003)
Booting from ROM...
iPXE (PCI 00:03.0) starting execution...ok
iPXE initialising devices...ok



iPXE 1.21.1+git-20220113.fbbdc3926-0ubuntu2 -- Open Source Network Boot Firmware
 -- https://ipxe.org
Features: DNS HTTP HTTPS iSCSI NFS TFTP VLAN AoE ELF MBOOT PXE bzImage Menu PXEX
T

net0: 52:54:00:12:34:56 using 82540em on 0000:00:03.0 (Ethernet) [open]
  [Link:up, TX:0 TXE:0 RX:0 RXE:0]
Configuring (net0 52:54:00:12:34:56)...... ok
net0: 10.0.2.15/255.255.255.0 gw 10.0.2.2
net0: fec0::5054:ff:fe12:3456/64 gw fe80::2
net0: fe80::5054:ff:fe12:3456/64
Nothing to boot: No such file or directory (https://ipxe.org/2d03e13b)
No more network devices

No bootable device.
_
```

Test Case 2: Docker Image Management

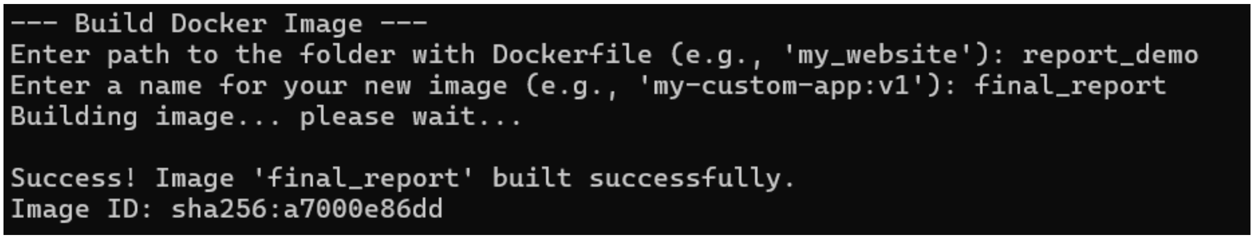
- **Objective:** Pull an image (`nginx`) and list it to verify persistence.
- **Result:** The system successfully connected to DockerHub, downloaded the layers, and displayed the image ID in the list.



```
--- Local Docker Images ---
ID: sha256:5c14a2f996 | Tags: my-python-test:v1
ID: sha256:fb01117203 | Tags: nginx:latest
ID: sha256:2d97f6910b | Tags: python:3.9-slim
ID: sha256:d4aaab6242 | Tags: hello-world:latest
----------------------------------------
```

Test Case 3: Custom Dockerfile Build

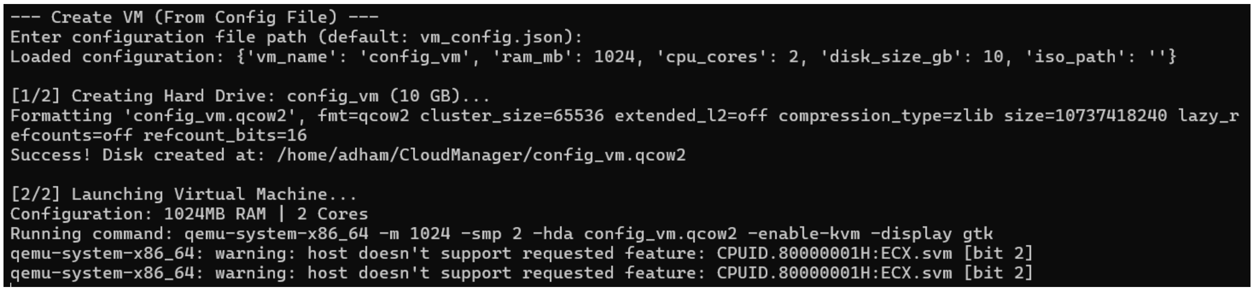
- **Objective:** Create a custom Dockerfile and build a new image from it.
- **Result:** The application generated the file structure and the Docker Engine successfully built the image tagged `my-python-test`.



```
--- Build Docker Image ---
Enter path to the folder with Dockerfile (e.g., 'my_website'): report_demo
Enter a name for your new image (e.g., 'my-custom-app:v1'): final_report
Building image... please wait...

Success! Image 'final_report' built successfully.
Image ID: sha256:a7000e86dd
```

Test Case 4: Configuration File Loading

- **Objective:** Load VM settings from `vm_config.json`.
- **Result:** The system correctly parsed the JSON file and launched the VM with the specific parameters defined in the file.



```
--- Create VM (From Config File) ---
Enter configuration file path (default: vm_config.json):
Loaded configuration: {'vm_name': 'config_vm', 'ram_mb': 1024, 'cpu_cores': 2, 'disk_size_gb': 10, 'iso_path': ''}

[1/2] Creating Hard Drive: config_vm (10 GB)...
Formatting 'config_vm.qcow2', fmt=qcow2 cluster_size=65536 extended_l2=off compression_type=zlib size=10737418240 lazy_r
efcounts=off refcount_bits=16
Success! Disk created at: /home/adham/CloudManager/config_vm.qcow2

[2/2] Launching Virtual Machine...
Configuration: 1024MB RAM | 2 Cores
Running command: qemu-system-x86_64 -m 1024 -smp 2 -hda config_vm.qcow2 -enable-kvm -display gtk
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bit 2]
qemu-system-x86_64: warning: host doesn't support requested feature: CPUID.80000001H:ECX.svm [bit 2]
```

5. Conclusion

The project successfully meets all outlined objectives. By leveraging Python's automation capabilities, we created a tool that simplifies the complex syntax of QEMU and Docker into a user-friendly menu. The system handles errors gracefully and provides a robust foundation for cloud resource management.