Project description
DevOps Infrastructure Provisioning & Configuration Automation Project

Overview

This is a rolling project that will evolve as new topics are learned. At this stage, the goal is to build the skeleton of an infrastructure provisioning tool. Future enhancements will integrate AWS and Terraform to create real resources. For now, the provisioning process is mocked to simulate infrastructure automation.

Project Objective

Develop a modular Python-based automation tool that simulates infrastructure provisioning and service configuration. The project should:

- Accept user inputs for defining virtual machines (VMs).
- Validate input using Python.
- Use classes and modular code structure.
- Automate service installation using Bash scripts.
- Implement proper logging and error handling.

Setup & Repository Initialization

- Create a GitHub repository for the project.

Define a clear project structure:

```
infra-automation/
|-- scripts/
|-- configs/
|-- logs/
|-- src/
|-- README.md
```

- Set up a virtual environment for Python dependencies.
- Push the initial setup to GitHub.

Simulating Infrastructure Provisioning

Accepting User Input & Validation

- The tool should allow users to define machines dynamically.
- Input should be validated to ensure correctness.
- Store configurations in a JSON file (configs/instances.json).
- Use a library like jsonschema or pydantic for validation.

Task:

- Create a function to prompt users for input.
- Implement validation to reject invalid inputs.
- Store the collected data in instances.json.

Hint: Think about what fields a VM requires (e.g., name, OS, CPU, RAM) and ensure users enter valid values.

Building a Modular Python Application

Using Classes and Imports

- Create a Machine class in src/machine.py.
- This class should:
  - Store machine details.
  - Provide a method to return a dictionary representation.
  - Include logging functionality for machine creation.

Task:

- Implement a basic class structure.
- Refactor existing script logic to use the class.
- Ensure the main script (infra_simulator.py) interacts with the class correctly.

Refactor Exercise: If an initial script exists without classes, refactor it into an object-oriented design.

Automating Service Installation with Bash

Integrating Bash Scripts for Configuration

- Write a Bash script to install and configure a basic service (e.g., Nginx).
- Modify the Python script to execute the Bash script after provisioning.
- Use Python's subprocess module to run the script.
- Ensure proper error handling in both Python and Bash.

Task:

- Write a Bash script to automate service installation.
- Modify the Python script to call the Bash script.
- Add error handling for potential failures.

Hint: The Bash script should check if the package is already installed before installing it.

Logging & Error Handling

Enhancing the Logging System

- Implement logging in both Python and Bash scripts.
- Write logs to a dedicated file (logs/provisioning.log).
- Ensure logs capture:
  - When provisioning starts and ends.
  - Any errors encountered.
  - Success messages for completed actions.

Task:

- Add logging to Python scripts using the logging module.
- Ensure Bash scripts output meaningful logs.
- Improve error handling by catching and logging exceptions.

Finalizing & Documentation

Wrap-Up & Project Documentation

- Update README.md to include:
  - Project overview and objectives.
  - Setup and execution instructions.
  - Example expected output.
- Ensure code is clean and modular.
- Push the final version to GitHub.

Task:

- Review the project for structure and readability.
- Add comments where necessary.
- Ensure documentation is clear and concise.

Next Steps & Future Enhancements

This project is designed to evolve. Once AWS and Terraform are introduced, extend the project to:

- Replace the mock provisioning with real AWS instances.
- Automate infrastructure management using Terraform.
- Introduce more complex services and configurations.

For now, focus on building a well-structured, modular, and professional Python application that serves as a foundation for future enhancements.

Code Help and Example
Helper Code Snippets for - DevOps Infrastructure Provisioning Project

```
## **1. User Input & Validation**
# Sample structure to collect user input and validate it.
import json

def get_user_input():
    machines = []
    while True:
        name = input("Enter machine name (or 'done' to finish): ")
        if name.lower() == 'done':
            break
        os = input("Enter OS (Ubuntu/CentOS): ")
        cpu = input("Enter CPU (e.g., 2vCPU): ")
        ram = input("Enter RAM (e.g., 4GB): ")

        instance_data = {"name": name, "os": os, "cpu": cpu, "ram":
ram}

        # Validate input (to be implemented by the student)
        # Example: validate_instance_input(instance_data)
        machines.append(instance_data)

    return machines

# Save to file
instances = get_user_input()
with open("configs/instances.json", "w") as f:
    json.dump(instances, f, indent=4)

## **2. Class Structure for Machine Representation**
# Basic class structure for managing machine objects
class Machine:
    def __init__(self, name, os, cpu, ram):
        self.name = name
        self.os = os
        self.cpu = cpu
        self.ram = ram

    def to_dict(self):
        return {
            "name": self.name,
            "os": self.os,
            "cpu": self.cpu,
```

```
            "ram": self.ram
        }

    def log_creation(self):
        print(f"Provisioning {self.name}: {self.os}, {self.cpu}, {s
elf.ram}")

# Example usage:
machine = Machine("web-server", "Ubuntu", "2vCPU", "4GB")
machine.log_creation()

## **3. Running Bash Scripts from Python**
import subprocess

def run_setup_script():
    try:
        subprocess.run(["bash", "scripts/setup_nginx.sh"],
check=True)
        print("[INFO] Nginx installation completed.")
    except subprocess.CalledProcessError as e:
        print(f"[ERROR] Failed to install Nginx: {e}")

# Example call:
# run_setup_script()

## **4. Logging & Error Handling**
import logging

# Configure logging
logging.basicConfig(
    filename='logs/provisioning.log',
    level=logging.INFO,
    format='%(asctime)s - %(levelname)s - %(message)s'
)

def log_message(message, level="info"):
    if level == "error":
        logging.error(message)
    else:
        logging.info(message)
    print(message)

# Example usage:
log_message("Provisioning started.")
log_message("Provisioning failed due to network issue.",
level="error")

## **5. Structuring the Project for Modularity**
# Example of how to split the project into multiple files and
import them.
# src/__init__.py (empty file to make this a package)

# src/machine.py
```

```python
class Machine:
    def __init__(self, name, os, cpu, ram):
        self.name = name
        self.os = os
        self.cpu = cpu
        self.ram = ram

    def to_dict(self):
        return {
            "name": self.name,
            "os": self.os,
            "cpu": self.cpu,
            "ram": self.ram
        }

# src/logger.py
import logging

def setup_logging():
    logging.basicConfig(
        filename='logs/provisioning.log',
        level=logging.INFO,
        format='%(asctime)s - %(levelname)s - %(message)s'
    )
    return logging.getLogger()

logger = setup_logging()

# Example of using the logger
# logger.info("This is a log message")

# main.py (Example integration)
from src.machine import Machine
from src.logger import logger

def main():
    machine = Machine("db-server", "CentOS", "4vCPU", "8GB")
    logger.info(f"Provisioning {machine.name}")
    print(machine.to_dict())

if __name__ == "__main__":
    main()

# **Next Steps**
# - Extend validation logic
# - Integrate more services
# - Replace mock provisioning with AWS/Terraform when those
modules are learned
```