**<INFO-8995-24F-Container and Orchestration>**

**<Andy Chow>**

**<Assignment-4>**

**<Group8>**

**<Varun Gundemoni, Ajay Panchal>**

**Objective**

Create a complete container-based artifact building/uploading solution to GitHub like a production deployment pipeline without having access to official systems. This plan describes all the steps along with the tools, potential challenges, and demonstration setup for Part I of this assignment.

**Approach:**

1. **Core Process Overview**

**Code Compilation:**

* Compile source code against a binary in a containerized environment.
* Build Dependency - everything should be installed in the container.

**Binary Packaging:**

* Zip or tar the created binary and make it deployable.

**Uploading to GitHub:**

* Upload to an appropriate repository using GitHub CLI or API, authenticating through personal access tokens securely stored in that environment.

**2. Simulation Strategy**

**Build Environment:**

* Use Docker Compose to define multiple services in a single configuration and deploy them together.
* Use separate Docker files to install the relevant libraries (e.g., GCC, Python, Nodejs) for their builds.

**GitHub Repository:**

* Create a mock repository to mimic the upload.
* Use a personal repository for testing.

**CI/CD Pipeline:**

* Simulate an automated pipeline using a shell script or GitHub Actions.
* Containers would run the jobs for building and uploading the binaries.

**Cost Considerations**:

* Use open-source tools (e.g., Docker, GitHub CLI) to minimize costs.
* Ensure the system is lightweight and efficient to run on local machines.

**3. Tools and Technologies:**

**Docker:**

For creating containerized build environments. Mainly to guarantee consistent and isolated environments that remove "works on my machine" challenges and give flexibility to simulate actual-world conditions.

**Justification**: Docker is available on all operating systems, it has tools and an extensive ecosystem, and is easy to set up. Podman is again an alternative container technology, but it is much more practical here because of the integration with Docker Compose and its mature documentation.

**Why are we not choosing Podman and Going with Docker**? Integration with Docker Compose is easy. Podman Compose, on the other hand, seems to be less evolved and may not offer certain functionalities.

It offers extensive community support, tutorials, and third-party elements into the Docker ecosystem.

**Docker Compose:**

Using it to orchestrate multiple containers into a single configuration helps manage dependencies and interactions between services, such as build and upload services, for more scalable and reproducible setups.

**Justification:** Docker Compose provides an excellent declarative model for multi-container applications and allows all configurations to be specified using a YAML-based file for easy usage and maintenance.

**GitHub:**

The most reliable and most-used code repository. GitHub API also allows seamless uploading of binaries while integrating well into CI/CD pipeline workflows.

**Justification:** GitHub has proven to be a suitable place for hosting both repositories and binaries. Its inherent CI/CD tools, such as GitHub Actions, are built to integrate easily into the workflow. GitLab has been equipped with many similar features, but the wider user base, better third-party integration, and more comprehensive documentation make GitHub a more feasible tool for this project.

**Why GitHub Over GitLab?**:

* GitHub Actions is natively integrated, reducing complexity and setup time compared to GitLab CI/CD.
* GitHub has better compatibility with external CI/CD tools, ensuring flexibility in future enhancements.

**Shell Scripting:**

It automates by just bringing things like container orchestration, authentication, and file uploads together. It gives a lightweight, flexible solution to carry local automation without fancy and additional tools.

**Justification:** Shell scripting is absolutely simple, lightweight, and widely supported on almost all platforms. Although Python scripting is more feature-rich and offers advanced logic handling, shell scripting is enough for simple basic automation tasks.

**So Why we choose Shell Scripting Rather Than Python Scripting?**

* **Efficiency**: Shell scripts are actually faster to write and execute such basic functionality as running Docker commands; managing files; and using environment variables.
* **Simplicity:** Shell scripts are much more straightforward and standalone for simple workflows without third-party libraries required to maintain.

**GitHub Actions:**

An optional but powerful feature for mimicking CI/CD pipelines. With native integration into GitHub repositories, it can easily serve as an example for demonstrating automated workflows.

**Justification:** GitHub has native integration with its own GitHub Action services, and that makes configuration simple along with reducing setup time. Although powerful, the Jenkins CI/CD tool has its challenges in creating and maintaining servers of its own, as with has an overhead cost.

**Why We Choose GitHub Actions Over Jenkins?**

* Simplicity: Set up and manage workflows using GitHub Actions, as it's integrated with your GitHub repositories rather than looking at Jenkins.
* Cost: Many free actions by GitHub are available for public repositories and hold free quota for private repositories, whereas Jenkins requires maintaining separate infrastructure.

**4.Detailed Steps**

1. **Establish a Local Development Environment**

* Create Mock Application with Sample Source Code(e.g C/Python program).

1. **Create Docker Images:**

Create Dockerfile for:

* Compiling Source Code
* Upload Binary to GitHub

1. **Define Docker Compose Configuration:**

Create a docker-compose.yml file that describes the service used for example:

* Build service: Compiling binary.
* Upload Service: Handles GitHub upload

1. **Shell Script Automation:**

That includes steps to:

* Build, and bring up the services with Docker Compose.
* Compile Binary.
* Authenticate GitHub.
* Upload Binary to the Repository.

1. **Test the System:**

Run Docker Compose configuration to validate the process from end to end.

1. **Document:**

* Document well areas in which the environment cannot be fully mimicked, such as real deployment environments with restrictions.
* Describe possible blockers, such as API limits, dependency conflicts, or authentication issues, and suggest ways to mitigate these issues

**5. Potential Challenges and Solutions**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  | | --- | --- | | **Challenge** | **Proposed Solution** | | GitHub API Authentication | Use secure storage for tokens and configure authentication within the container. | | Mimicking Real Environments | Simulate only key parts (e.g., build and upload) with clear documentation on unaddressed gaps. | | Network Issues with GitHub Upload | Test with retry mechanisms and provide fallback error messages for clarity. | |

**6. Best Practices**

* Use descriptive Docker files and script comments.
* Provide a single command to execute the demo process.
* Secure sensitive credentials using environment variables.
* Use lightweight containers for faster builds.

This plan will serve to demonstrate the feasibility of a containerized build and upload system for later refinements in Part II.