**Assignment 4**

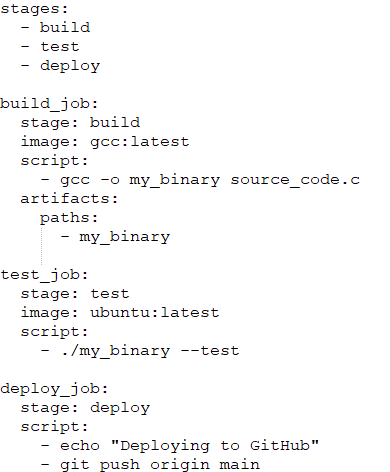
Group 6- Punit Darira, Jagjeet Singh

Cloud Development and Operations, Conestoga College  
  
INFO8995-24F-Sec1-Container and Orchestration

Course Instructor- Andy Chow

Assignment Due Date-4th December 2024

Proposed plan involves creating a DevOps pipeline using GitLab CI/CD with self-hosted VM runners. This setup will allow you to build and deploy a compiled binary, simulating an environment where you don't have access to official environments. Here's a detailed plan and the potential limitations of this approach.

1. **GitLab CI/CD Pipeline Configuration**
   * **.gitlab-ci.yml File**: Every codebase should have this file in the root. This file will define the stages, jobs, and the Docker images to be used for building the codebase. Each job will specify the required image and any necessary scripts to compile the binary.
   * **Example Configuration**:  
     
2. **Self-Hosted VM Runners**
   * **Installation**: We would need to Install GitLab Runner on your VMs. Each VM can be configured to handle specific hardware requirements (e.g., x86\_64, ARM64) if we want to divide workload among different VMs. Since the build would be running in containers, it doesn’t matter on which VM the container gets created.
   * **Registration**: Register each runner with your GitLab instance, specifying tags to identify the hardware capabilities. For example, a runner on an ARM64 machine could have the tag arm64.
3. **iOS Development**
   * **macOS Runner**: For iOS development, we will need a macOS runner. This runner must be a physical Mac machine or a macOS VM running on Apple hardware.
   * **Setup**: Install GitLab Runner on the macOS machine and register it with your GitLab instance. Use tags to identify it as the iOS/macOS runner.

**Advantages**

1. **Persistent Containers**: Unlike GitLab managed runners, self-hosted runners can maintain persistent containers, i.e., we won’t need to start up containers each time, which can save time and resources. We can queue builds for containers that are already running.
2. **Flexibility**: Self-hosted runners allows customization of environment to meet specific needs, such as using different hardware architectures or specific software versions.
3. **Cost Efficiency**: If we have existing hardware, using self-hosted runners can be more cost-effective than using cloud-based managed runners.
4. **Control**: We have full control over the runners, including security settings, resource allocation, and maintenance schedules.
5. **Performance**: Self-hosted runners can be optimized for performance, potentially leading to faster build and deployment times compared to shared managed runners.

**Limitations**

1. **Builds which require hardware**: Code bases like iOS development, firmware for microcontrollers, code bases which rely on GPU features require physical hardware for the build. In this cases, it may be hard to setup GitLab runners on these hardwares.
2. **Complexity**: Setting up and maintaining multiple runners for different architectures adds complexity to your CI/CD pipeline.
3. **Scalability**: While self-hosted runners provide flexibility, scaling them can be challenging compared to using cloud-based runners.
4. **Maintenance**: Regularly updates to runners to avoid compatibility issues and security vulnerabilities.

**Conclusion**

Using GitLab CI/CD with self-hosted VM runners is a powerful way to simulate a production-like environment for our DevOps pipeline. Jenkins self hosted runners are also an option which. Since Jenkins has a bigger community and larger plugin ecosystem.