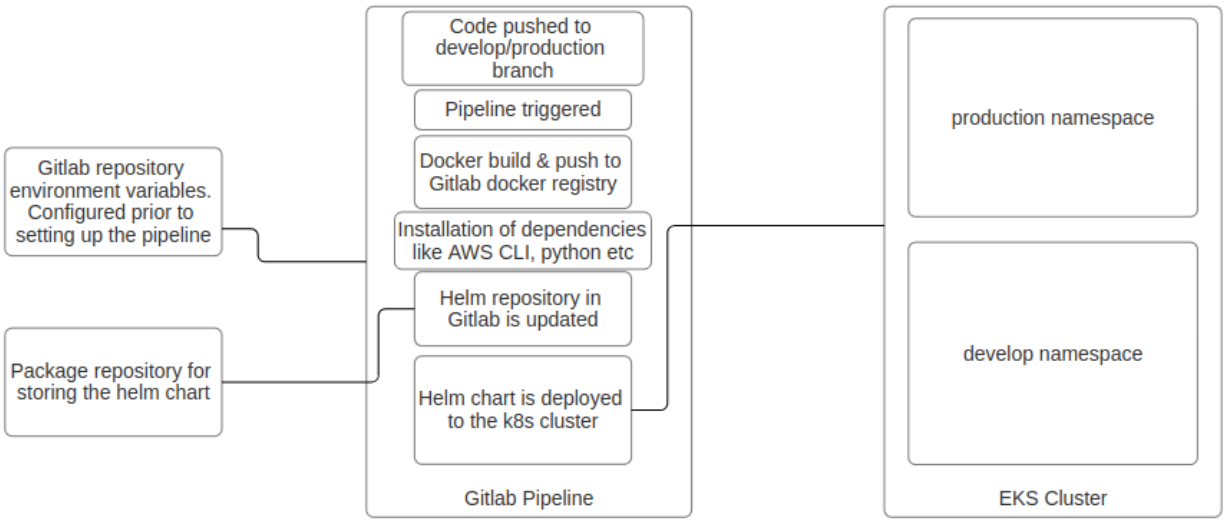
Detailed explanation of the architecture and components of your CI/CD pipeline

The Gitlab CI/CD Pipeline consists of the following **components**:
Gitlab Runner: The server on which the pipeline executes.
Gitlab Repository: To contain all the files, i-e, Dockerfile, source code etc.
Dockerfile: Through which docker container will be built
Helm charts: Kubernetes templates for deployment, service, and ingress objects.
Kuberntes cluster: On which our application will be deployed.
Environment Variables: For storing the access key and secret.
Package Registry: For storing the helm chart.

The Gitlab CI/CD Pipeline will have the following **architecture**:



Step by step instructions for configuring GitLab CI/CD, Docker, and Kubernetes to orchestrate the pipeline

Prerequisites:

1. Navigate to Gitlab repository > Settings > CI/CD
2. On the CI/CD settings page, add the following variables:
   AWS_KEY_ID
   AWS_SECRET_KEY
   AWS_REGION
3. Register a Gitlab Runner or use Gitlab's self-managed Runner.
   3a. To use your own runner, navigate to Gitlab repository > Settings > CI/CD
   3b. Under Runners, choose New project runner and follow the instructions
4. For this task, we will be using Gitlab's Docker registry. Navigate to Gitlab repository > Deploy > Container Registry and see the authentication process. We will be using Gitlab's predefined environment variables for authenticating with the registry.

5. For storing our helm chart, we will be using Gitlab's package registry. We can push our helm chart using one the approach mentioned here

Pipeline:
The pipeline uses two docker images for two stages. For build_and_push stage, `docker` image is used, and for deploy stage, alpine/helm image is used.

The first stage uses docker-in-docker service to authenticate, build, and push the docker image to the gitlab docker registry.

The second stage first installs dependencies like python3, awscli etc and then executes the main script. It adds a helm repository, which is created as a prerequisite. Then it uses aws-cli to authenticate with the EKS cluster using the variables saved in step 2 of prerequisites. Lastly it uses the helm upgrade command to update the deployed helm chart. The newly built image name is supplied as a parameter to this command. The branch name is used as the namespace in which the deployment takes place.

Description of the rollback strategy and how it would be triggered and executed

We can have a separate pipeline, which would run some cases. If any of the cases fails, helm rollback command can be used to trigger a rollback.

This pipeline can be triggered as a downstream pipeline by the above-mentioned pipeline.

Discussion on how you would handle versioning and dependencies management within the application

For versioning of our application docker images, we can either use git tags or we can use commit hashes. But for this task, branch name + timestamp is used.

For dependency management, we can use package.json file or requirements.txt file depending on the language our application is written in.

Any additional tools or techniques you would use to enhance the reliability, scalability, and observability of the CI/CD pipeline

We can implement **retry** to achieve reliability. We can follow this official documentation to achieve that.
We can have **parallelism** to achieve scalability. We can follow this documentation to achieve that.
We can integrate **OpenTelementry, ELK stack,** or any other monitoring tool to achieve observability of our CI/CD pipeline.