



**Linnæus University**  
Sweden

## Deployment Pipeline - Project Report

# Continuous Delivery

*2DV611 - Group 3*



*Author:* Elias Frigård, Karl Ekberg, Joel Salo,  
Fredrik Ljungner, Erik Haga  
*Supervisor:* Francis Palma, Diego Perez  
*Semester:* HT21



## Table of contents

<b>The Application .....</b>	<b>3</b>
<b>Pipeline Setup .....</b>	<b>4</b>
<i>Overview .....</i>	<i>4</i>
<i>How to run the pipeline?.....</i>	<i>4</i>
<i>Gitlab Variables .....</i>	<i>4</i>
<b>The pipeline.....</b>	<b>5</b>
<i>Pipeline workflow .....</i>	<i>5</i>
<i>Pipeline stages .....</i>	<i>6</i>
Stage 1: Dependencies .....	6
Stage 2: Test.....	7
Stage 3: Build.....	8
Stage 4: Staging.....	10
Stage 5: Production .....	11
Stage 6: Destroy (optional) .....	12
<i>How does it all work? .....</i>	<i>13</i>
<i>Pipeline Draft .....</i>	<i>14</i>
<i>Using Gitlab's Pipeline UI.....</i>	<i>15</i>
Dependencies between stages .....	16
<i>Artifacts.....</i>	<i>17</i>
Test results .....	17
<i>Notifications .....</i>	<i>18</i>
<b>Glossary.....</b>	<b>20</b>
Zero downtime .....	20
Blue/Green deployment.....	20
Configuration drift .....	21
<b>Further Reading.....</b>	<b>21</b>



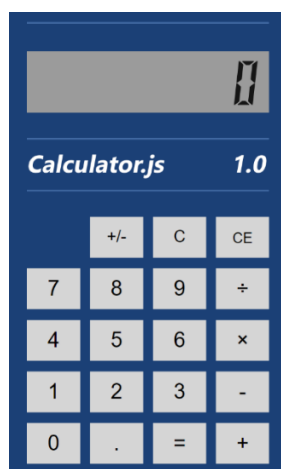
## The Application

The application that this team has been working on is a basic **calculator application**. The calculator is a simple, “classic” one. Meaning that it can only add, subtract, divide and multiply two operands. The application is built in HTML and CSS with the logic being handled by JavaScript and Node.js. The application came bundled with unit tests written using Mocha. The team has been developing acceptance, smoke, and integration tests to test the application even further.

The application in its original state did not provide any functionality for persistence, such as saving or retrieving previous calculations. Thus, to have a “fully fledged” application with all the required layers needed to perform integration testing, the development team needed to modify the existing application and add persistence to it. Over time the team has been working on adding this functionality to the application in small increments and today the app has support for saving history.

The application is not yet hosted on any static domain or address, but by running the Gitlab pipeline you can set the application up on your own OpenStack cloud environment. Within said pipeline, numerous types of systems and applications work in unison to deploy the application into staging and production environments. The systems that work together are Terraform, Ansible, Docker and Kubernetes. We will delve deeper into these systems as the documentation progresses.

The repository of this project, including pipeline and application can be found [here](#).





## Pipeline Setup

### Overview

The CI/CD configuration for the pipeline lies within a specific file that the pipeline consumes. This file is named `.gitlab-ci.yml` and can be found in the root directory of the project. The file follows a YAML syntax, meaning that it has specific rules about indentation which you can read more about [here](#).

The main `.gitlab-ci.yml` also includes references to specific, decoupled configuration YAML files that are specific for each stage of the pipeline and can be found under the `.gitlab/ci/` directory.

### How to run the pipeline?

There are two ways to run the pipeline:

1. Clone the project repository and push your code by running the commands `git add`, `git commit -m '<message>'`, and `git push`. The pipeline will then consume the code within the repository.
2. Alternatively, you can make changes to the code or the pipeline-file directly in Gitlab. The editor for the pipeline-file can be found in the left menu, under CI/CD and “Editor”. The pipeline also triggers if there’s a change within the code and GitLab also provides a built-in IDE which can be used to edit code. However, we will not cover that functionality here.

### Gitlab Variables

Note that as a new user, your credentials to Openstack must be added to Gitlab. This is since Openstack environments are private, and we do not share these. You can add your credentials by going to *Settings* → *CI/CD* → *Variables* → *Expand*.

Currently, these are the variables that need to be set:

- `ANSIBLE_KEY_<name>` (Your SSH-key from Openstack)
- `OS_KEY_NAME_<name>`
- `OS_KEY_PAIR_<name>`



- `OS_PASSWORD_<name>`
- `OS_TENANT_ID_<name>`
- `OS_USERNAME_<name>`

Every parameter that must be set in your Gitlab variables can be found in Openstack. In the top right corner of Openstack you can click on your username and fetch the information from “Openstack RC File 2”. The use of these variables will be explained in the ‘Workflow’ section.

Read more about GitLab variables [here](#).

## The pipeline

### Pipeline workflow

The pipeline currently uses the *workflow* keyword with a set of rules that help us run the pipeline with different variables depending on the person who commits the changes. These variables are linked to each person’s OpenStack account. This was created to be able to run separate environments, since we were not provided with a unique shared OpenStack environment.

The code that utilizes the different variables looks like the example below. The pipeline runs jobs with a different configuration depending on the user that initialized the commit. In this example, the credentials of Fredrik are used, and the jobs will be run against his Openstack / CSCloud environment.

```
1 workflow:
2   rules:
3     - if: $GITLAB_USER_NAME =~ /^Fredrik/
4       variables:
5         OS_KEY_NAME: ${OS_KEY_NAME_FREDRIK}
6         OS_KEY_PAIR: ${OS_KEY_PAIR_FREDRIK}
7         OS_PASSWORD: ${OS_PASSWORD_FREDRIK}
8         OS_TENANT_ID: ${OS_TENANT_ID_FREDRIK}
9         OS_TENANT_NAME: ${OS_TENANT_NAME_FREDRIK}
10        OS_USERNAME: ${OS_USERNAME_FREDRIK}
11        ANSIBLE_KEY: ${ANSIBLE_KEY_FREDRIK}
12      when: always
```



The workflow rules can be found in the `.gitlab/ci/_workflow.gitlab-ci.yml` file. As a new developer, you will need to add a new rule for your own variables, following the existent format.

Read more about workflow rules [here](#).

## Pipeline stages

The pipeline consists of 5 stages with an additional 6th optional stage, where each stage represents a milestone within the pipeline.

### Stage 1: Dependencies

Within the “Dependencies”-stage there’s four different jobs:

- **Node-modules:** Downloads a node image and install the necessary dependencies for the project (basically running *npm install*). After this is done, the pre-work for Terraform runs.
- **Terraform-files:** Uses templates to create different terraform “main” and “out” files for the corresponding blue and green environments. This helps avoid configuration drift issues.
- **Terraform-init:** Initializes the Terraform script. It exports the variables set inside the tfvars-file inside the project before it runs *terraform init*.
- **Terraform-plan:** Runs *terraform plan* which creates an execution plan, this is to allow Terraform to make use of state, meaning that already existing objects which have not been changed, will not run.

#### *Troubleshooting errors:*

This stage rarely fails. If it does, it’s usually because of some error with installing the node modules, perhaps the path to the script is wrong (for example wrong path



to the source folder). By looking closer at the failed job, you will be able to find out more information of what went wrong.

```
25 $ npm --prefix ./src install ./src
26 /usr/bin/bash: line 105: npm: command not found
```

If there's something wrong with Terraform you will be able to view the terraform error messages here as well. Often, if Terraform fails in some way, there's most likely a syntax error within the Terraform template files. Other than that, there might be an issue with the connection to Openstack, however this is quite unlikely.

## Stage 2: Test

*Test-app* runs the stage *Test* where the unit-tests run. When the unit tests are completed, they produce an artifact containing the test results (artifacts will be discussed in-depth later on). However, the test results can also be viewed in *CI/CD*, *Pipelines* and inside each pipeline, there's a tab called "Tests". It's important to remember that if one test fails, the whole pipeline shuts down. This stage depends on the previous stage. A more in-depth description of the different dependencies will be discussed more in-depth later on.

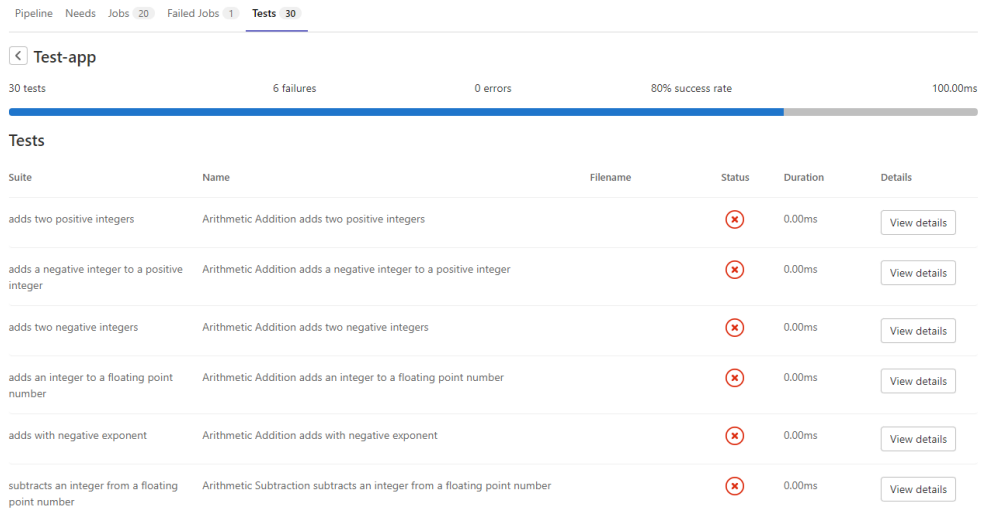
### *Troubleshooting errors:*

If the test stage fails, it's most likely due to some issue with the unit tests. Much like the potential pathing error above, the path in the configuration file might be wrong, or an error might be caused by the fact that one or more tests has actually failed. If a test fails, you may view said test inside the failed job, or you may go to the "Test"-tab inside the stage.

---

Pipeline Needs Jobs 6 Tests 30

---



Above we can see an example of failing unit tests. These errors were provoked by changing the logic of the add function to subtract instead. In this case, we can see that this change resulted in six failing unit tests and their respective output. By clicking “view details” you can get more information about the failure and what part of the code caused the tests to fail. A pipeline that fails unit tests will not proceed to any later stage and deployment will be aborted. The developer will be notified using one or more of the notification methods we will discuss later in this documentation.

### Stage 3: Build

The *Build* stage is quite extensive and triggers after the test stage has been completed without any issues. The build stage is responsible for creating the environment that will later be utilized by the pipeline. This stage includes the following jobs:

- **Build-image:** Creates an image for the application that in the Dependencies stage was built. Other than creating the image, this job also tags said image and pushes the image to GitLab’s built-in Docker image repository.
- **Build-environment:** Creates the environment that will be used. This job creates both the blue and the green environment (which will be covered





later on). This job utilizes the pre-existing Terraform scripts in order to properly set the environment up for use. Within this job IP-addresses also get noted to text files for use later on.

- **Blue-cluster** and **Green-cluster**: These two jobs set up Kubernetes clusters for both green and blue environments using Terraform and Ansible scripts.
- **Configure-runner (manual)**: Configures the runner created in the build-environment job. This job is manual due to the fact that we should not reconfigure the cluster each time the pipeline runs.

Please note that at least one runner must be active 24/7 in order for the stages to run. The runners can be found under Settings → CI / CD → Runners:

#### Available specific runners

<div><div></div><div>#428 (7a38aq8a)</div><div>eh223ub-2dv517</div></div>	<div><div></div><div></div><div>Remove runner</div></div>
<div><div></div><div>#427 (exqyynRc)</div><div>js225fg-2dv517</div></div>	<div><div></div><div></div><div>Remove runner</div></div>
<div><div></div><div>#426 (RC1zzGaa)</div><div>ef222xf-2dv517</div></div>	<div><div></div><div></div><div>Remove runner</div></div>
<div><div></div><div>#414 (K4Nx1pKo)</div><div>fl222ai-2dv517</div></div>	<div><div></div><div></div><div>Remove runner</div></div>

#### *Troubleshooting errors:*

Since there's five jobs in this stage, there's a risk for various errors occurring. Primarily, the biggest risks are new configurations Terraform, however as these are intact as of right now, those errors would most likely only occur if a developer were to make changes to the provisioning or configuration management code.

Other than that, the error that may be most common is with Openstack itself which unfortunately is beyond the reach and control of our team. Errors concerning Openstack should be reported to the cloud team.



## Stage 4: Staging

Staging is the stage where a new version of the application first gets deployed to an environment. This stage triggers an SSH-connection in order to apply the Ansible playbook to deploy the application to the cloud. The stage creates a staging environment that is virtually identical to the deployment environment. By utilizing the green / blue deployment strategy we alternate between the staging and deployment environments. These two environments switch roles in acting as the publicly deployed application, and the one used for staging and testing.

Apart from deploying to the staging environment, this stage also runs the final tests. Those tests run in the order of Smoke tests → Integration tests → Acceptance tests. The stage includes the following jobs:

- **Deploy-app-staging:** This is where the Ansible-playbook “deploy\_app\_new.yaml” runs. This is a “master script” which runs each Ansible playbook needed in order to configure the environment properly. This job only runs the corresponding Ansible playbooks to the current testing environment.
- **Smoke-test:** The smoke test runs only to see if there’s “smoke coming out of the application”. Essentially it tries to connect to the application (curling) and it expects the application to return status code 200 (OK).
- **Integration-test:** When the smoke test is done, it passes on to the integration tests to make sure that the database connection is intact. Apart from establishing the connection, this test also creates an entry in the database, verifies that this entry is correct and deletes that entry. If this stage passes, the application has a proper connection to the database.
- **Acceptance-test:** Includes a test suite that runs scraping tests using [Puppeteer](#). This final testing stage makes sure that the application works as intended, essentially doing calculations using the UI and looks for the correct result.



## *Troubleshooting errors:*

The issues that may reside here are configuration issues with Ansible. If this happens, have a look at the job to find the error message from Ansible. Also, if Openstack goes down during this stage it would cause an issue as well.

Other errors that might occur here are errors in testing, meaning that a test has failed. If this happens, have a look at the job that has failed to deduce what test has gone wrong. The image below is a simple example of output from running acceptance tests. Here we can see that eight tests pass and two fail. By looking at the log of the job we can find more information about the failing unit tests, what provoked them and hints on how to fix these errors.

```
Acceptance tests
MongoDB Connected
/ (Home Page)
  ✓ should have the correct page title
  ✓ should display number when clicked (57ms)
  ✓ should display decimal number (102ms)
  ✓ should clear the display when C is clicked
  ✓ should display result of adding two numbers (643ms)
  ✓ should display result of subtracting two numbers (627ms)
  ✓ should display result of dividing two numbers (667ms)
  ✓ should display result of multiplying two numbers (629ms)
  1) calculator history should display last calculation
  2) calculator history should display last calculation (after reloading page)
8 passing (4s)
2 failing
```

## **Stage 5: Production**

This stage shifts the staging environment to be the production environment and vice versa shifts the current production environment to be the staging environment. If we make a change that we do not like, we can use the “roll-back” manual function to reverse the switch in environments. In practice, the floating IP of the staging environment is switched with the IP of the deployment environment to make staging and production environments accessible on their respective IP addresses regardless of which green or blue instance environments are used for the current deployment.



- **Deploy (manual):** The first time the pipeline runs, the application gets deployed to the blue environment. After this, each time the pipeline runs and you choose to deploy from the staging environment, the IP-addresses shift. If blue is the current production environment, the IP-address will shift to the green load balancer, creating zero downtime.
- **Roll-back (manual):** This job rolls back the environments, meaning that it shifts the IP-addresses back to its previous state.

### *Troubleshooting errors:*

This stage merely deploys a new deployment environment, and this stage depends on previous stages, this stage should not throw any errors corresponding to the environment. If this stage were to fail, it's most likely due to changes in the Terraform scripts which concerns deployment only. However, since this stage swaps IP between the different environments, Openstack may fail if the connection to Openstack fails or that Openstack fails by itself.

### **Stage 6: Destroy (optional)**

In order to not have to tear down the environment manually in CSCloud we have created this optional stage. This stage automatically destroys the environment. The user can choose to only destroy the blue or the green environment, the runner or everything within the cloud.

No developers (only maintainers or above) have access to this stage, so you don't have to worry about a new dev accidentally destroying the environment.

### *Troubleshooting errors:*

Errors that might arise here are purely with Openstack and should be reported to the cloud team. Also note that if you destroy your runner, you should delete it in the Gitlab interface as well.



## How does it all work?

To get a good grip on the pipeline and how it works you may follow the flowchart which is attached. Essentially, the pipeline triggers when a commit has been made to the repository. When this happens, the stages run in the specified order (listed above in the “Stages” section) going from Dependencies → Test → Build → Staging → Production. Finally, there’s one last stage which is completely optional, that stage is “Destroy”.

Note that Deploy is also optional and it is a manual stage. However, Deploy is essential since it deploys the application to the live environment.

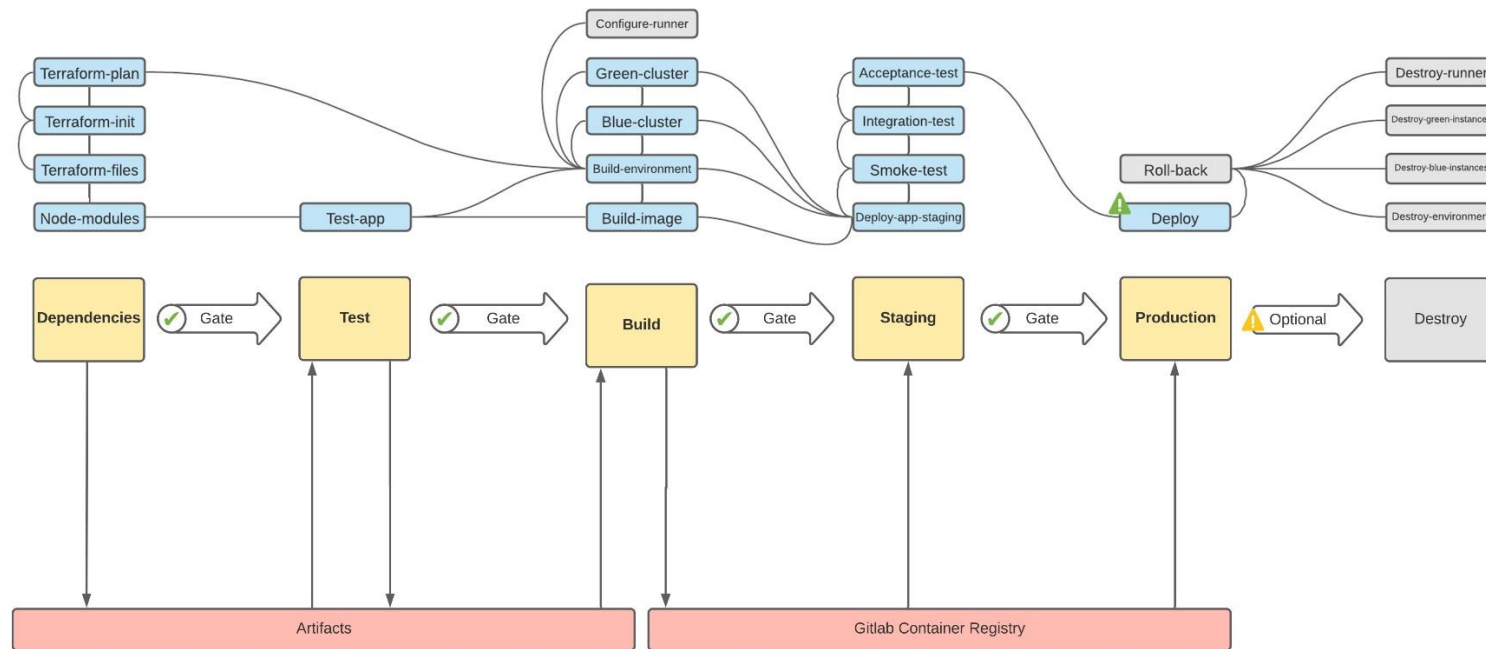
When speaking of a live environment we refer to the environment that is open to the outside world. This is made possible since we utilize a pattern known as “blue/green deployment”. There’s always one environment (either blue or green) live towards outside users.

To be able to continue to develop and test the application, all new features get deployed to the other environment (if blue is live, the “new application” gets deployed to green). When the pipeline passes for the green environment, the user can simply push a button and switch the environment. Now the green environment is the one that is live, meanwhile the blue environment is the testing environment.



## Pipeline Draft


The following is a visual draft of the pipeline in its current form. It visualizes the different stages, jobs, artifacts, and dependencies used by the different pipeline elements. This draft has changed significantly from the initial draft and continues to change as the pipeline development process grows and evolves. To compare with the original draft, please see the original project plan.






## Using Gitlab's Pipeline UI

Each stage and each job may succeed or fail. Each stage results in logs from the stages that have been run. When a stage has been completed, you can find out what the stage has been doing by going to CI/CD → Pipeline. There you can see the state of the pipeline (eg. failed, passed, canceled):

 **failed**


[#78328](#)

---

 **passed**

[#78324](#)


---


 **canceled**


[#78323](#)


You can click the pipeline that you wish to view, and then, inside the pipeline, you can click on each job included in each of the stages. For example, inside the “Dependencies” stage you may view information about the jobs within that stage:


**Dependencies**


 Node-modu...





 Terraform-fi...



 Terraform-init



 Terraform-...





By clicking on “Node-module” you can find extensive information about the job. If the job succeeded, you would see “Job succeeded” in green letters at the very bottom.

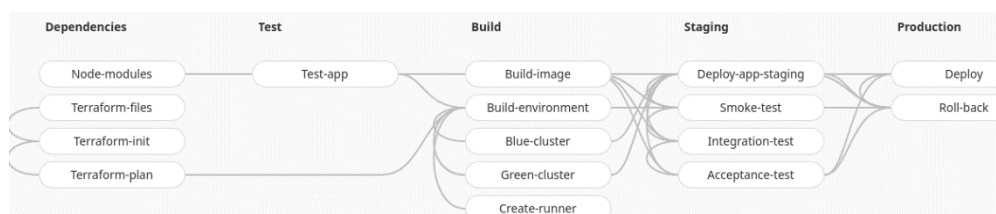
```
46  Job succeeded
```

However, there may be jobs that fail. In that case, you will be presented with various error messages that show what went wrong and at the very bottom you will find an error-line.

```
353  ERROR: Job failed: exit code 1
```

## Dependencies between stages

Inside the editor you can also click “Visualize” to get a clearer image of the dependencies between the stages. By specifying the keyword “needs” such as “needs: [“job”]” you can create dependency arrows inside this tab.



If you want to create a “needs”-visualization you can add the keyword to the job that should depend on another job such as:

```
Smoke-test:
  stage: Staging
  image: curlimages/curl:7.80.0
  needs: ["Node-modules", "Build-environment", "Deploy-app-staging"]
  environment:
```





## Artifacts

“Artifacts” are files stored on the Gitlab server after a job is executed. Artifacts can also be set to be destroyed after a specific amount of time.

Some stages produce “artifacts”. This is output generated from the stages. We will later have a look at the testing artifacts which are the test results produced by the stage. To not have to reinstall dependencies for each job we also use artifacts to keep node modules. Artifacts are also used to keep Terraform-plan files and it is also used to store information such as IP-addresses for Ansible.

Note that some stages require artifacts from previous stages to be able to run. The keyword “dependencies” specifies if the stage is depending on artifacts generated by another stage. One very important artifact is the test results for the unit tests which will be covered below.

Read more about GitLab job artifacts in the [official documentation](#).

## Test results

When the unit tests finish inside the testing stage it will produce an artifact. This artifact does only show the tests that have been done and its outcome.

The outcome can be found under the pipeline that you just ran. Inside this test tab you can view each test’s artifact, which in this case is the test result. The tests either pass or fail. Note that if even one test fails, the whole pipeline shuts down and you will get notified either via the Discord hook, or via email (explained more in-depth in the coming section).



< Test-app

30 tests0 failures0 errors100% success rate140.00ms

Tests					
Suite	Name	Filename	Status	Duration	Details
rejects missing operation	Arithmetic Validation rejects missing operation		✓	38.00ms	<a href="#">View details</a>
adds two positive integers	Arithmetic Addition adds two positive integers		✓	15.00ms	<a href="#">View details</a>
divides by zero	Arithmetic Division divides by zero		✓	7.00ms	<a href="#">View details</a>
adds zero to an integer	Arithmetic Addition adds zero to an integer		✓	5.00ms	<a href="#">View details</a>
adds a negative integer to a positive integer	Arithmetic Addition adds a negative integer to a positive integer		✓	4.00ms	<a href="#">View details</a>
divides by zero expect null	Arithmetic Division divides by zero expect null		✓	4.00ms	<a href="#">View details</a>
rejects invalid operation	Arithmetic Validation rejects invalid operation		✓	3.00ms	<a href="#">View details</a>

## Notifications

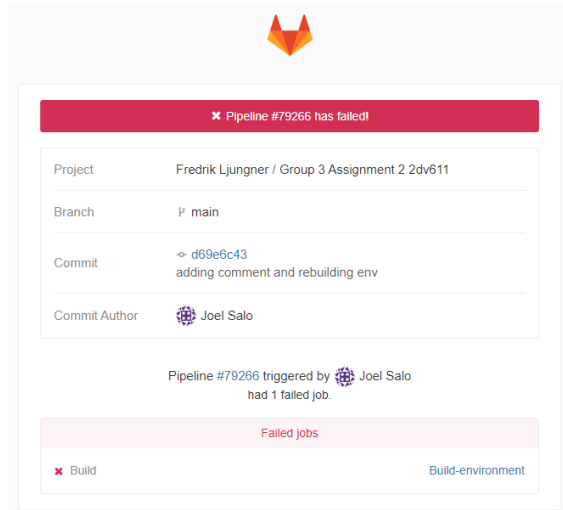
As developers we want to be notified if certain events occur. Gitlab supports several kinds of notifications (based on different events) through several different kinds of channels, allowing us to customize the desired workflow.

For our project we decided that we want to be notified if the pipeline fails at any point in time and that the main communication channel should be *email*. In addition to receiving an email notifying that the pipeline has failed (example below) we also implemented a *webhook* to also receive notifications in Discord. The Discord notifications should trigger on *any* event regarding the pipeline in order to enable developers to quickly get an overview of the current activity in the project.

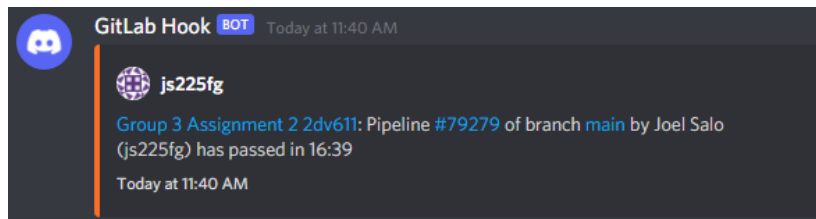
More information on GitLab's notifications can be found [here](#).



*E-mail:*



*Discord:*



For more information about GitLab webhooks, please read more in the GitLab documentation [here](#).



## Glossary

### **Zero downtime**

Zero downtime is a method created and used in order to minimize the downtime in the application when a new release has been created and passed through the pipeline, and eventually deployed. We reach zero downtime by applying a well-known pattern known as Blue/Green deployment (which will be covered below).

By having two separate environments, the only downtime will occur when Openstack is changing the IP-address from one environment to the other. However, there's no real zero downtime as this takes a moment for Openstack to execute, the downtime is however severely reduced down to just a couple of seconds.

### **Blue/Green deployment**

Blue/Green deployment is an analogy of two different environments. Blue and green can in practice be named whatever, blue and green is the most common naming convention for this pattern. Visualize blue and green as two separate environments where the infrastructure and core applications (Docker, Kubernetes etc) are identical.

The application will first get staged and deployed to the blue environment. Whenever a change occurs, the changed application will get deployed to the green environment.

When a change gets committed, the change will be deployed to the staging environment in the "current non-live environment" (green environment if blue environment is production, blue environment if green environment is production). The only environment visible to outside users of the application is the production environment, meaning that the staging environment can be used as a testing environment where more tests such as manual user acceptance tests can be conducted.



This staging environment also has a public IP-address only known to the developers. However, as soon as the developers are happy with the new changes, you can push “deploy” and the IP-addresses will get shifted in Openstack. Meaning that the current production environment will become unknown to the outside world, and the current testing environment takes its place.

## **Configuration drift**

Configuration drift refers to an environment progressively over time that may start to differ from the other environment. This is an unwanted stage and may be caused by inconsistent configurations between the stages.

This is why the job “Terraform-files” exists. This job inside the Dependencies stage exists solely to create two identical environments, meaning that Terraform utilizes a template provided to create two identical environments generated by Ansible Configuration drift.

## Further Reading

The following section contains some useful links that can be important to understanding how the application works and where to learn some of the tools involved with the development and administration process.

[Ansible Documentation](#)

[Terraform Documentation](#)

[Kubernetes Documentation](#)

[Docker Documentation](#)

[Gitlab Documentation](#)