GSoC'22 Proposal - LitmusChaos

Terraform Provider for Litmus

- by **Pratik Singh**

Organization Name: **Cloud Native Compute Foundation**

Project Name: LitmusChaos

Candidate Name: Pratik Singh (@kitarp29)

CNCF Project Description: Here

LitmusChaos Issue: Here

Mentor(s): Vedant Shrotria (@Jonsy13), Raj Babu Das (@rajdas98), Adarsh Kumar (@Adarshkumar14)

Expected project size: 350 Hours

Difficulty: Medium

Abstract

Develop a terraform provider (along with the documentation) on top of the helm provider to provision litmus with the following operations

- Install chaos center on k8s
- Install chaos agent via helm
- Add chaoshub to ChaosCenter via API provider
- Run chaos workflows via API provider

Table of Contents

I. Summary

LitmusChaos is a tool that enables SREs and developers to find a weakness in their deployments by running chaos experiments in the staging environment to find and detect bugs and vulnerabilities. The Litmus Portal provides a graphical user interface and is the console from which to manage, monitor, and send events around the chaos workflows.

We need to develop a Terraform provider to provision LitmusChaos. Publish it as a provider at Terraform Registry. Utilize the API provider to run chaos workflows and add Chaoshub to ChaosCenter.

II. Description of the Problem

In this project, we will develop a Terraform provider on top of the Helm provider to provision Litmus with the different operations. This provider will enable developers to install chaos center on k8s and chaos agent using Terraform scripts. Also, help to add chaoshub to ChaosCenter and run chaos workflows via API provider. We will need to add tests before publishing the terraform provider. The provider documentation should be written and published as a Terraform provider at Terraform Registry.

2.1.1 Terraform

Terraform is a tool for building, changing, and versioning infrastructure safely and efficiently. It is a declarative language that describes the desired state of infrastructure as a set of configuration files. Terraform is used to provision infrastructure, change infrastructure, and manage infrastructure. Terraform uses **Terraform providers** to assign resources based on Terraform scripts. Terraform Scripts uses a declarative language developed by Hashicorp known as *Hashicorp Configuration Language*. There are multiple use cases for this tool

- Multi-Cloud Deployment
- Application Infrastructure Deployment, Scaling, and Monitoring Tools
- Self-Service Clusters
- Kubernetes on different providers
- PaaS Application Setup

> Pictorial representation of how Terraform works

2.1.2 Terraform Providers

Terraform providers are a set of tools that Terraform can use to provision infrastructure. Terraform providers are written in **Go** and are written in a way that is easy to use and maintain. All of the official providers can be found on the Hashicorp registry (Here) and can be used by developers to build and maintain infrastructure using Terraform Scripts.

2.2 Current Approach

Currently, LitmusChaos has no official Terraform Provider. If an SRE or DevOps Engineer has to deploy a resource with Litmuschaos using terraform scripts. The options are:

- Use Helm Provider to install Litmuschaos on k8s and chaos center using terraform scripts.
- Use Kubernetes Provider to do the same.
- Manually install LitmusChaos on the resources.

There might be some other options as well. But most of them require manual intervention. None of them are suitable for SREs to Create, Read, Update and Delete LitmusChaos-based resources using Terraform Scripts. There is a need to develop a Terraform provider to provision LitmusChaos on Kubernetes using scripts.

III. **Proposed Solution**

3.1 **Making a Terraform Provider**

In Terraform, a **Provider** is the logical abstraction of an upstream API. Terraform supports a plugin model, and all providers are actually plugins. Plugins are distributed as Go binaries. Although technically it is possible to write a plugin in another language, we will be going ahead with Go for this project.

There are a few reasons to use Terraform plugins. One of them is to provide a way to write Terraform scripts that are not supported by the official Terraform providers. The stats on Terraform registry are huge. A few official providers have over 10 million downloads per week. Terraform allows resources to begin managing existing infrastructure components once they migrate to Terraform. Lots of operators will bring their existing infrastructure under Terraform's control.

> *Rationale behind using Golang.*
> Almost all Terraform plugins are written in Go. Most of the code in LitmusChaos runs on Golang. Golang is great for scaling and concurrent programming.

Implementation Details

Any Terraform Provider has four most basic parts:
i. The Provider Schema
ii. The Executable file
iii. Defining Resources
iv. CRUD Operations

3.1.1 **The Provider Schema**

Schema is used to define attributes and their behaviors, using a high-level package exposed by Terraform Core named `schema.Providers` , Resources, and Provisioners all contain schemas and Terraform Core uses them to produce a plan and apply executions based on the behaviors described.

This schema is a collection of key-value pairs of schema elements. The attributes a user can specify in their Terraform scripts. The keys are strings, and the values are `schema.Schema` structs that define the behavior.

In our solution, we will use the `schema.Provider` type to define the provider's properties. We can have multiple properties as per the requirements. The minimum requirement is to have a `kubeconfig` property. It will tell the provider where to find the Kubernetes cluster. The `kubeconfig` property is a string that is the path to the kubeconfig file. There are other properties that can be specified. We may also have a struct of all the necessary parameters for the provider schema.

The schema takes care of the arguments that are supplied to it within the Terraform script. The schema will also be used to validate the arguments supplied to it. But the schema is not an executable file. It is a simple collection of key-value pairs.

An example of proposed schema is:

```
import (
    "github.com/hashicorp/terraform-plugin-sdk/v2/helper/schema"
)

func Provider() *schema.Provider {
    return &schema.Provider{

        ResourcesMap: map[string]*schema.Resource{
            "example_server": resourceServer(),
        },

        Schema: map[string]*schema.Schema{
            "kubeconfig": {
                Type:        schema.TypeString,
                Required:    true,
            },
            "kubeconfig_content": {
                Type:        schema.TypeString,
                Optional:    true,
                Default:     "",
            }
        },

        ConfigureFunc: func(d *schema.ResourceData) (interface{}, error) {
            return &config{
                kubeconfig:        d.Get("kubeconfig").(string),
                kubeconfigContent: d.Get("kubeconfig_content").(string),
                    }, nil
                },
    }
}
```

3.1.2 **The Executable file**

Go requires a `main.go` file, which is the default executable when the binary is built. The `main.go` file is the entry point of the program. It can be the only file that is required to be present in a plugin. Since Terraform plugins are distributed as Go binaries, it is important to define this entry-point file. We can also include the schema in the `main.go` file. But to maintain the simplicity and readability of the code, we will define the schema in a separate file.

A sample `main.go` file for proposed terraform provider:

```
package main
import (
        "github.com/hashicorp/terraform-plugin-sdk/plugin"
        "github.com/hashicorp/terraform-plugin-sdk/terraform"
)
func main() {
        plugin.Serve(&plugin.ServeOpts{
                ProviderFunc: func() terraform.ResourceProvider {
                        return Provider()
                },
        })
}
```

This establishes the main function to produce a valid, executable Go binary. The contents of the main function consume Terraform's plugin library. This library deals with all the communication between Terraform core and the plugin.

We initiate the module, download and install the dependencies. Next, build the plugin using the Go toolchain. Verify the results by executing.

```
go build -o litmuschaos-provider ./main.go
```

Next, we have to define the resources.

3.2 **Defining Resources with CRUD**

A resource is a component of the terraform provider. Defining the ability to **create, read, update,** and **delete** resources is an important part of creating Terraform Providers. Terraform lifecycle operations are mapped into API operations. And resources are the final outcome of the Terraform script.

> As an example, the Google provider supports `google_compute_instance` and `google_compute_address`.

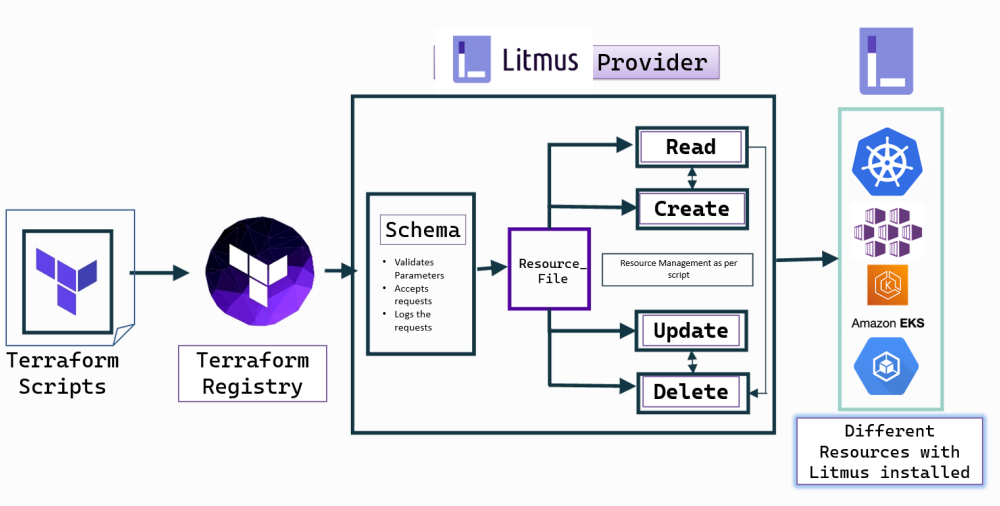
General convention is that Terraform providers put each resource in their own file, named after the resource, prefixed with `resource_`.

A sample resource file looks like this:

```
package main
import (
        "github.com/hashicorp/terraform-plugin-sdk/helper/schema"
)
func resourceServer() *schema.Resource {

        return &schema.Resource{
                Create: resourceServerCreate,
                Read:   resourceServerRead,
                Update: resourceServerUpdate,
```

```
                Delete: resourceServerDelete,
                },
        }
}
```

We will be defining these functions to explain the logic behind the provisioning of the resource as per the Terraform Scripts. Helm will be called in these functions to initiate and manage litmuschaos on the resources.



3.2.1 **Create**

The function will be called in the case of `Terraform apply`. It will create the `litmus` namespace and install the helm chart in it. There is a pre-requisite that Helm needs to be installed on the cluster. We could have another validation step to check if the helm is installed or not.

We will be using the `client-go` library to create the namespace.

```
nsSpec := &v1.Namespace{ObjectMeta: metav1.ObjectMeta{Name: "litmus"}}
_, err := clientset.Core().Namespaces().Create(nsSpec)
```

We will use the official Helm dependency to install the `litmus` chart. This is a small snippet that will be a part of **Create** :

```
chart, err := loader.Load("https://litmuschaos.github.io/litmus-helm/")
 iCli := action.NewInstall(actionConfig)
    iCli.Namespace = "litmus"
    rel, err := iCli.Run(chart, nil)
    if err != nil {
        panic(err)
    }
    fmt.Println("Successfully installed LitmusChaos")
```

3.2.2 **Read**

This function will read the `litmus` namespace and check if the `litmus` namespace is created or not. We will accomplish this using the `client-go` library.

```
_, err := clientset.Core().Namespaces().Get("litmus", metav1.GetOptions{})

    if err != nil {
        panic(err)
    }
    fmt.Println("Litmus Namespace exists")
```

This function will act as a validation step. If the namespace is not created, it will return an error. It might call back the **Create** function in the case of an error. We might also check for the other resources in the `litmus` namespace. We can have a check to see if all the important Kubernetes components for Chaos are present or not. We can have checks for these resources:

- chaos-exporter
- chaos-operator-metrics
- litmuschaos-frontend-service
- litmuschaos-server-service
- litmusportal-frontend-service

This might also be called after the steps of the **Delete** function to validate the destruction of all resources.

3.2.3 **Update**

This function will also have a similar format of dependencies. The function will have multiple parameters as per the use case. The function would manage the components in the `litmus` namespace.

If needed we can also have dependencies other than `client-go` to help us run *kubectl* commands directly like in a terminal. We can call **Read** Function internally to measure the difference between the current configs and the requested changes.

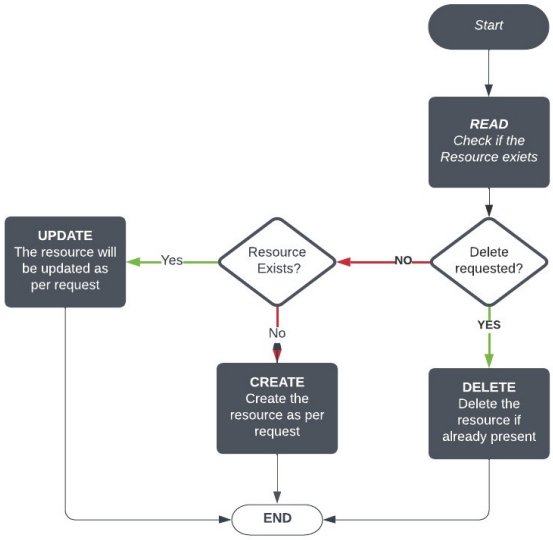
A checker for the difference between the requested resources and the existing resources. This will have a logging system setup to have a history of the changes. This will help the developers to understand the changes.

This is a user-specific function and has to be discussed.

3.2.4 **Delete**

This function will be used in the case of a `terraform destroy` operation. It will delete the `litmus` namespace and all the resources in it. It will delete all the components in the `litmus` namespace. We could use either `client-go` or `kubectl` to delete the resources. We will callback the **Read** function to check if the namespace and other components are deleted or not.

> Flowchart representing Logic for CRUD Operations



3.3 **Using API Provider to add chaoshub and run Chaos workflows**

The Litmus Portal API Provider is based on GraphQL. It can be used to create and manage chaosworkflows and more. Documentation can be found Here.

Before running any type of mutation on the GraphQL API, we need to authenticate the user. We will be creating a JWT Token first by hitting the `Auth server`. The `POST` request will have the following parameters in the Body:

```
{
  "username": "admin",
  "password": "litmus"
}
```

We will be using this JWT Token to authenticate the user on GraphQL API. Once the user is authenticated, we can use the API Provider to run and add chaos on our cluster. We can follow the steps from Here.

3.4 **Testing and Publishing Terraform Provider**

Publishing the Terraform Provider is a very important step. Terraform has very well-written steps to help developers publish providers Here. We will need to complete two more steps to publish the Terraform Provider. Any terraform provider needs these two things to get published:

- Test Cases
- Documentation

3.4.1 **Test Cases**

We will need to have code to test the terraform provider. We will be referring to this Documentation Here to create the test cases.

Terraform's `resource` package offers a method Test(), accepting two parameters that act as the entry point to Terraform's acceptance test framework. The basic two parameters of these functions are:

- The standard `*testing.T` struct from Golang's Testing package. Here
- `TestCase`, a Go struct that developers use to set up the acceptance tests. Here

This section will require a lot of work. We will be creating the test cases for the terraform provider. We will have to handle error handling in these files. Also, will take care of partial states.

TestCase offers several fields for developers to add to customize and validate each test, defined below.

- **IsUnitTest** : It allows a test to run regardless of the TF_ACC environment variable.
- **PreCheck** : It is commonly used to verify that required values exist for testing, such as environment variables containing test keys that are used to configure the Provider or Resource under test.
- **Providers**: In our case, this is the most important test case. Only the Providers included in this map are loaded during the test, so any Provider included in a configuration file for testing must be represented in this map or the test will fail during initialization.
- **Steps**: Basic tests typically contain one to two steps, to verify the resource can be created and subsequently updated, depending on the properties of the resource. In general, simply creating/destroying tests will only need one step.
- **CheckDestroy**: It is called after all test steps have been run and Terraform has run destroy on the remaining state.

Here is a sample test case for the provider:

```
package litmusTest

func TestLitmus(t *testing.T) {
  var clusterBefore, clusterAfter litmusTest.Widget

    resource.Test(t, resource.TestCase{

    // check for  litmus namespace and the resources within
    PreCheck:     func() { SimpleBaiscTests(t) },

    // check for litmusProvider
    Providers:  litmuschaos-provider,

    // destroys all the rest tests, executes at the end
    CheckDestroy: testAccCheckExampleResourceDestroy,

    // we define each step for testing
    Steps: []resource.TestStep{
      {
        Config: TestClusterResource(clusterBefore),
        Check: resource.ComposeTestCheckFunc(
          test_for_litmus_exist(&ClusterBefore),
        ),
      },
      {
        Config: TestClusterResource(clusterAfter),
```

```
          Check: resource.ComposeTestCheckFunc(
            testAccCheckExampleResourceExists(&clusterAfter),
          ),
        },
      },
    })
}
```

> We might add more functions and steps as per the requirements.

3.4.2 **Documentation**

Terraform helps to develop and maintain the documentation for our provider with each release. There is a tool that helps to generate the documentation for the terraform provider. We will be using this tool to generate the documentation for the terraform provider.

More on this can be found in the documentation at Terraform Here.

3.4.3 **Publishing Provider**

We can publish and share a provider by signing into the Registry using our GitHub account and following a few additional steps. Only after publishing the provider on Terraform registry will help Developers utilise it in their Terraform scripts. More info is in the documentation : Here

3.5 **Dependencies**

There are official Golang dependencies of Terraform that will use in this:

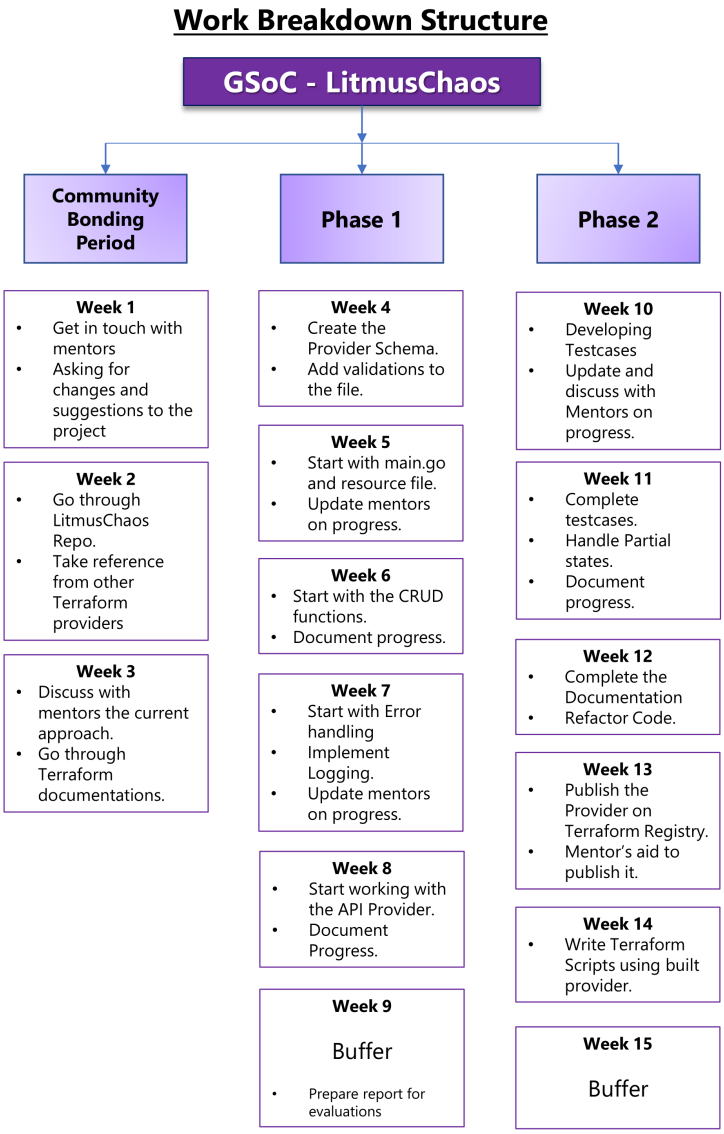
```
// Terraform Dependencies

github.com/hashicorp/terraform-plugin-sdk/v2/helper/schema
github.com/hashicorp/terraform-plugin-sdk/plugin
github.com/hashicorp/terraform-plugin-sdk/terraform


// Client-Go depenedencies

k8s.io/client-go/kubernetes
k8s.io/kubernetes/pkg/api/v1
k8s.io/api/core/v1
k8s.io/apimachinery/pkg/util/runtime
k8s.io/apimachinery/pkg/apis/meta/v1
k8s.io/client-go/kubernetes
k8s.io/client-go/tools/clientcmd

// Helm Dependencies

helm.sh/helm/v3/pkg/action
helm.sh/helm/v3/pkg/chart/loader
```

```
helm.sh/helm/v3/pkg/kube

// Logging of Errors and other activities

github.com/sirupsen/logrus
```

IV. **Deliverables and Timeline**

0. **Before Community Bonding** (April 19- May 19)

- Objective is to push a good number of Pull Requests to the LitmsChaos GitHub repository.
- Completing the already assigned tasks. I have worked on some features on the project, will try to complete it as soon as possible.
- Take a course on Terraform.Refer and study the GitHub repos of various official terraform providers.



1. **Community Bonding Period** (May 20 - June 12)

1.1 Week 1:

- Contact the Litmuschaos Developers (Raj, Vedant, Ajeesh, and others), and try to get in touch with them.
- Get in touch with my project mentor, introduce myself, and work out a good time to communicate (taking into account their schedule).
- Ask the mentor for any suggestions/changes to the project. The project proposal reflects my knowledge of writing, there may be essential amendments necessary and it would be better to make them early.

1.2 Week 2:

- Go through the Litmuschaos codebase again. Make notes of all the parameters needed to install the project on a Kubernetes cluster.
- Go through some of the examples of terraform providers. Discuss the different approaches taken by other providers took, so it is essential to get familiar with them.
- Get in touch with other GSoCers, both within the CNCF community and outside.

1.3 Week 3:

- Terraform in itself is a pretty detailed tool for Infrastructure as a Code. To make a Terraform provider would require me to learn and implement the tools as well. So going through the Terraform documentation is essential.
- Discover the current approaches to installing LitmusChaos on a Cluster (local/hosted). Want to look into options of:
    - Write Terraform script to install Litmuschaos using Helm Provider.
    - Use Kubernetes provider to install LitmusChaos using helm charts.
- Once tested the other methods, we will keep a note of the shortcomings in each approach. We will be using them to improve the overall performance of the project later.

2. **Phase 1** (June 13 - July 29)

2.1 Week 4:

- Initialising the project by creating the *Provider Schema*. Discuss and finalize the parameters user will provide using the Terraform Scripts.
- We will declare the validation tests in this file only. We can return different errors based on the inputs.
- Give the mentors an update on my progress so far and take their feedback.

2.2 Week 5:

- Start with the executible file (`main.go`). The *provider* will be called in this file.
- Start with the basic implementations of the *resource* file. Completing atleast one of the functions of CRUD (preferably *Read* function).
- Check the integartion of files so far. Develop and test the first build.

2.3 Week 6:

- Complete all the main functions of CRUD. The preferred order will be Read and Create first.
- Moving ahead will be the Delete and Update functions.
- Discuss with Mentors the working of the functions. Give an update on the work so far.

2.4 Week 7:

- Complete the build with CRUD operations, if any are left from last week.
- Implementing Logging features in the provider.
- Add error handling to the functions.
- Check for partial states.

2.5 Week 8:

- Start with initiating Chaos from the API Provider. Learning the other aspects of the API Provider.
- Adding chaoshub to ChaosCenter via API provider.
- Discuss with Mentors the working of the functions. Give an update on the work so far.

2.6 Week 9 and rest of Phase 1 :

- *Buffer* for any pending tasks.
- Prepare a report for evaluations.

3. **Phase 2** (July 25 - September 12)

3.1 Week 10:

- Developing and creating testcases for the provider.
- Making Testcses aligned with the exceptions and errors in the functions.
- Discuss with mentors on the progress. Starting with testing on different inputs.

3.2 Week 11:

- Complete the Testcases for the provider.
- Handle situations of Partial states.
- Document the progress so far.

3.3 Week 12:

- Complete the Documentation.
- Refactor the Code.
- Making sure to complete the functions and implementation before publishing.
- Discuss with mentors on the project. Get started with the next phase.

3.4 Week 13:

- Publish the provider on Terraform Provider.
- This phase will be needing help from the mentors as well.

3.5 Week 14:

- Make Terraform Scripts consuming the built provider.
- Make one-click solution to install LitmusChaos on any Kubernetes Cluster.(EKS, GKE or Local hosted)
- Look into more tasks.

3.5 Week 15 and rest of Phase 2:

- *Buffer* for any pending tasks.

> The dates mentioned above are tentative considering the global pandemic.

V. **Relevant Skills and Experience**

Open-source enthusiast, I have contributed to various open-source projects based on MERN Stack and DevOps. I got selected as the LiFT Scholar by The Linux Foundation under the Open-Source Newbie Category in 2021.

5.1 **Skills**:

- **Golang** : Worked on production level code during my last Internship. Worked on some DevOps projects built on Golang.
- **Kubernetes** : Worked on Kubernetes via kubectl and client-go both.
- **AWS Certified** : Trained and certified to use several services of AWS.
- **Google Cloud** : I was the facilitator of GCP Cloud Training at my campus. I was trained by the employees of Google. Helped my community members to get certified.
- **Linux** : Currently pursuing Essentials of Linux System Administration (LFS201) Course as a part of the LiFT scholarship. I have been selected by The Linux Foundation under Open Source Newbie Category. Ubuntu is my daily driver.
- **Knowledge of Chaos Engineering** : While working during my Github Externship, I have worked on Chaos Engineering.

5.2 **Experience**:

- In my previous Internship with Juspay, I worked on Golang for an in-house project. Learned to write industry-level Golang code. Worked on Kubernetes Clusters hosted on AWS EKS.
- During my Github externship, I got my introduction to LitmusChaos. Worked over it for a small period of time. Trying to learn and contribute more.
- Learned about fundamentals of DevOps in Emerging Technologies Course offered by my university.
- As a Google Developer Student Club Lead, I have taken workshops to impart the knowledge of DevOps to my fellow students. Here

---

VI. **Willingness to take up tasks after primary task**

I am a student at my core still trying to learn Golang and DevOps. I will be glad to contribute to the project after the provider is published on Terraform Registry.

I would be interested to take up tasks related to the latest Chaos Cloud the company introduced. Moreover, would love to be able to write some CRDs for the ChaosHub.

VII. **Availability**

- I would expect around 10-11 Weeks for this project.
- I have no commitments for this summer and will be available to contribute to the project.
- I will be available to communicate via Slack and available via Internet (Video Conference, Audio Conference, etc).
- I would keep the Mentors in the loop regarding the on-goings of the project. Update mentors on a bi-weekly basis.
- During the majority of the Coding period, my college will be closed for summer break.

VIII. **Questions for Mentors**

- What are the parameters we are expecting to include in the Provider. Schema? The challenges faced in current approaches. It will help us increase the parameters for the success of the project.
- Only we will be building the Terraform Provider? We can also provide a one-click solution for the user to install Litmus on different Cloud providers using our Terraform scripts.

IX. **References**

- Terraform Documentation: Here
- LitmusChoas Documentation: Here
- Kubernetes Documentation: Here
- Client-go: Here
- Google Cloud Tutorials: Here
- Hashicorp Videos: Here
- Reffered to code of some Providers on Terrafrom Registry: Here

---

X. **Contact**

Name: Pratik Singh

Email: kitarpsinghrajpoot@gmail.com

LinkedIn: https://www.linkedin.com/in/kitarp29/

GitHub: https://github.com/kitarp29

Mobile Number: +91 987654321
