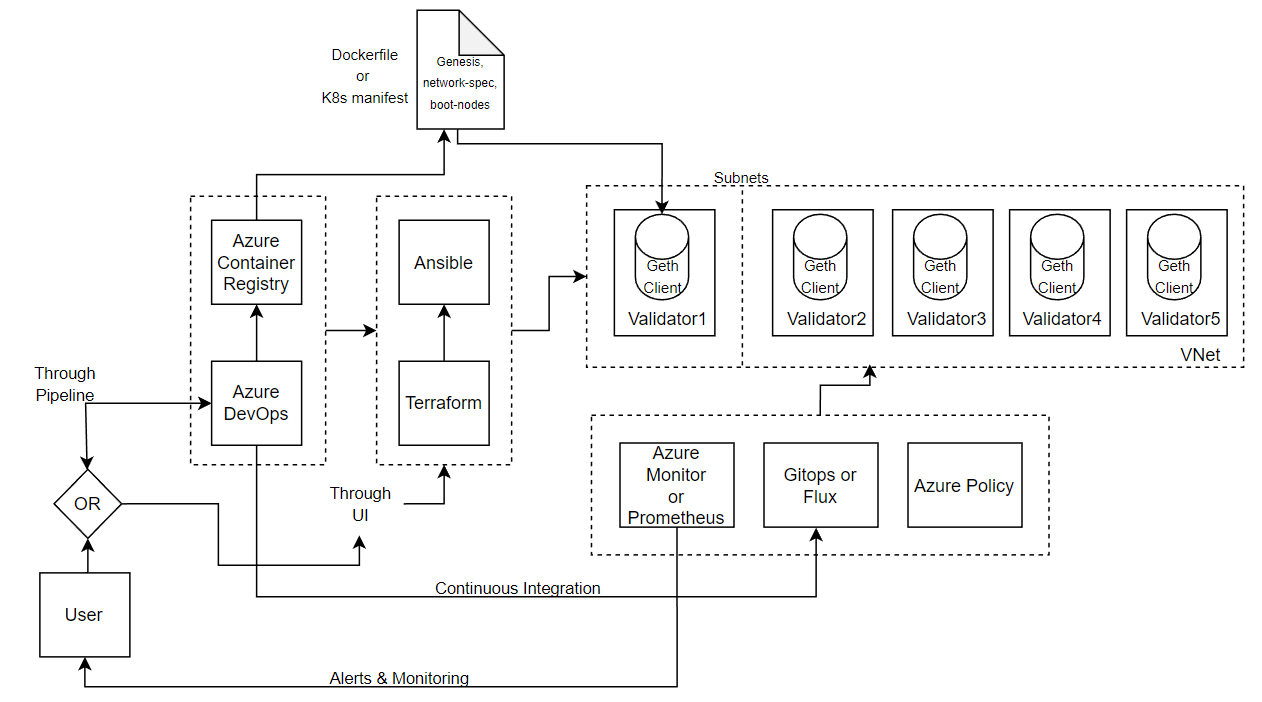
**Technical Design Document**

## **1. Introduction**

The purpose of this system is to automate the provisioning and setup of 5 validator nodes on a private Ethereum network using cloud services. Here the azure cloud has been used for the demonstration. The system leverages Terraform for infrastructure provisioning, Ansible for configuration management, and Azure services or open source tools such as Prometheus for monitoring and security.

## **2. System Architecture**

The system architecture consists of the following components:



### **2.1. Request Provisioning:**

The user initiates the provisioning process through a user interface or an Azure Pipeline. The user provides the desired specifications and count for the validator nodes.

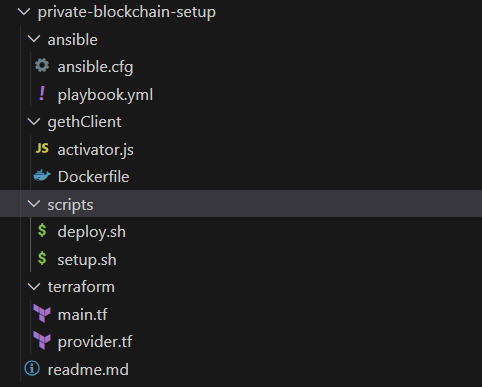
### **2.2. Automation Scripts**

The system includes several automation scripts to streamline the deployment process but the end user just needs to trigger the setup.sh, rest of the scripts will be processed internally:

* **setup.sh**: This script sets up the initial environment, installs dependencies, and triggers the deployment script.
* **deploy.sh**: This script orchestrates the provisioning, configuration, and deployment process. It calls Terraform and Ansible scripts.

### **2.3. Directory Structure**

The project directory structure is as follows:



### **2.4. Infrastructure Provisioning with Terraform**

The system uses Terraform for infrastructure provisioning. The Terraform configuration files are located in the **terraform/** directory:

* **main.tf**: Defines the resources to provision, such as VMs, VNets, subnets, NSGs, key-vaults and all other components etc., using the Azure provider.
* **provider.tf**: Specifies the provider configuration for Azure Resource Manager.

### **2.5. Configuration Management with Ansible**

Ansible is used for configuration management of the VMs. The Ansible playbook, located in the **ansible/** directory, performs the following tasks:

* Configures the VMs with the required internal services such as Docker or Kubernetes.
* Installs and configures the Geth client on each validator node.
* Handles additional configurations if necessary (e.g., starting the Geth client, activating validator functionality).

### **2.7. Image Building**

Following the configuration management step, the system proceeds with building the Docker or Kubernetes image of the Geth client. The image is built based on a Dockerfile or Kubernetes manifest file that specifies the required specifications, dependencies, and configurations.

* Docker: If Docker is chosen as the containerization technology, a Dockerfile is created. The Dockerfile defines the steps to build the Geth client image, including pulling necessary dependencies, copying configuration files, and setting up the entry point.
* Kubernetes: If Kubernetes is used, a Kubernetes manifest file (e.g., YAML) is created. The manifest file describes the specifications and configurations required for the Geth client deployment, such as container image, environment variables, and resource allocation.

The image building process can be performed locally or in a dedicated build environment. Once the image is built, it can be pushed to a container registry for further deployment and utilization.

#### **2.8. Docker/Kubernetes Deployment**

* Set up a container registry or image repository to store the pre-built Docker or Kubernetes image of the Geth client.
* Define the necessary Kubernetes manifest file or Dockerfile to build the image, specifying dependencies, configurations, and entry points.
* Push the built image to the container registry or image repository.
* Configure the deployment mechanism to pull the image and deploy it on each VM using Docker or Kubernetes.along with the required network ID, genesis block, boot nodes, and other network parameters that are included in the deployment configuration.

#### **2.9. Geth Client Initialization**

* Develop a startup script or leverage a service manager (e.g., systemd) to start the Geth client on each VM.
* Configure the startup script or service manager to run the Geth client with the appropriate parameters, including the necessary configuration files.

#### **2.10. Activating Validator Functionality**

* Implement a mechanism to trigger a staking transaction from the Ethereum accounts associated with each node. This could be achieved through a REST API endpoint that will be triggered once the synchronization with the network is complete.
* Set up the necessary infrastructure to interact with the Ethereum network and submit the staking transaction. Configure the system to automatically activate the validator functionality upon successful completion of the staking transaction.

#### **2.11. Resource Supervision**

* Determine the relevant metrics for monitoring the Geth client containers or pods (e.g., block propagation time, network latency, resource utilization).
* Configure Azure Monitor or Prometheus to collect and analyze the required metrics.
* Set up alerts or notifications based on predefined thresholds to notify the DevOps team in case of anomalies or significant deviations in the monitored metrics.
* Implement appropriate monitoring dashboards to visualize the collected metrics and provide insights into the health and performance of the validator nodes.

#### **2.12. Private Key Storage and Management - Key Vault**

* Set up an Azure Key Vault instance to securely store the private keys associated with the Ethereum accounts.
* Configure appropriate access policies and permissions to ensure that only authorized individuals or systems can access the keys.
* Implement mechanisms within the deployment scripts or configuration management tools to retrieve the private keys from Azure Key Vault during the setup or initialization process.
* Ensure that the private keys are properly protected and not exposed within the deployment scripts or configuration files.