VPN Implementation with Virtual Network Functions

Prateek Rath
IMT2022017

Mohit Naik
IMT2022076

Ananthakrishna K
IMT2022086

Contents

1 Introduction

A Virtual Network Function (VNF) is essentially a software implementation of a network function like a firewall, router, load balancer, VPN, etc. that runs on virtualized hardware instead of dedicated appliances, and can be scaled, orchestrated, and deployed using various DevOps tools and techniques. In this project, we have tried to implement a simple VPN along with a reverse proxy as virtual network functions, which we then deploy and monitor using various software tools.

2 Application

The basic idea is to have an application which runs on a protected network i.e. is not visible/accessible externally. Users who wish to access the application need to connect to the VPN first which is on the same network as the application, after which all requests to the application are forwarded through the VPN. We also add an Nginx reverse proxy at the front which takes all requests, filters out unwanted ones, and performs basic rate limiting.

2.1 Target App

This is a simple FastAPI application which is not externally exposed. It has just one functional endpoint `/status`, which returns an `OK` when requested.

2.2 VPN

The core VPN implementation. Users can connect to the VPN via providing their credentials, after which they are able to access the above application. When they disconnect, they are unable to access it again.

2.3 Nginx

We add a reverse proxy in front of the VPN for protection. It filters out any unwanted requests, and prevents the VPN from being bombarded via some rate limiting. It also becomes a central point for all requests so that the VPN and the target application can remain unexposed externally.

3 DevOps Components

We've used a modular architecture, with each of the above components as a separate service with its own Docker containers, with Kubernetes as the orchestration mechanism and Dockerhub as the container registry. We've also added tools like Prometheus and Grafana for monitoring metrics, and the ELK stack (Elasticsearch, Logstash, Kibana) with Filebeat, for logging. Furthermore, we've used Jenkins for continuous integration and deployment (CI/CD), and Ansible for infrastructure automation.

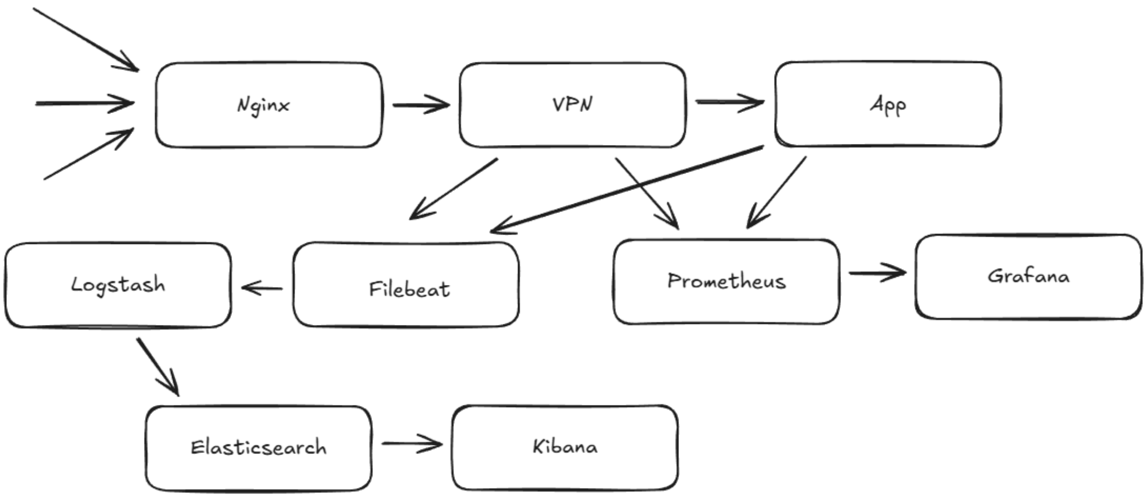

Figure 1: Architecture

3.1 Containerization and Orchestration

All the containerization and orchestration is present in the `k8s` folder. Broadly, we can divide our application's main functionality into:

- the nginx reverse proxy
- the vpn gateway where auth takes place
- the app itself which handles requests.

We have the following deployments:

- app
- elastic search
- grafana
- kibana
- nginx
- prometheus
- vpn

Some of the deployments such as logstash, prometheus etc use configmaps to store configurations that are then mounted in the deployed containers for use. Every deployment is exposed to the rest of the cluster via a service. Port forwarding helping us access those services from localhost.

Typically, filebeat is used as a daemonset and not a deployment, because it needs to read log paths from every node. Daemonsets automatically track cluster scaling i.e. when a new node joins, filebeat automatically starts tracking that node. Here since we are running on minikube (a single node cluster), it doesn't matter as much.

Each deployment folder is armed with a `kustomization.yaml` file that simply allows us to use the `-k` flag and apply all yaml files within a folder via a single command.

3.2 Monitoring

We've used Prometheus to scrape metrics from the app and vpn pods, and Grafana to visualize these metrics. The following metrics are scraped:

- Number of `/status` requests allowed by the vpn
- Number of api requests which fail due to invalid/expired JWT tokens

These metrics are sent to Grafana, which aggregates them and displays them as graphs/dashboards. The exact configuration for the dashboards has to be specified as JSON, which we have embedded into the k8s manifests for Grafana.

3.3 Logging

We use the Elastic Stack (ELK Stack)—Filebeat, Logstash, Elasticsearch, and Kibana—to collect, process, store, and visualize logs.

Filebeat runs on the application nodes and collects log files or container logs. It forwards these logs to Logstash or directly to Elasticsearch. Logstash is an optional but powerful processing layer where logs can be enriched, parsed, filtered, or transformed before they are indexed. The processed logs are then sent to Elasticsearch, which stores them in indices designed for fast search and analytics. Finally, Kibana sits on top of Elasticsearch and provides dashboards, charts, and real-time visualizations that help us understand system behavior, troubleshoot issues, and monitor application health.

This pipeline enables efficient log ingestion, structured indexing, and meaningful insights from logs in real time.

For this project we present 5 visualizations:

- Logs per container bar chart
- Logs over time line chart
- Error Logs v/s Normal Logs over time
- Top 5 pods with the most error logs
- Log Volume by Namespace Pie chart

To view these:

1. Deploy the application via the jenkins pipeline.
2. Switch to the jenkins user by running `sudo su - jenkins`
3. Run `kubectl port-forward svc/kibana 30090:5601 -n logging`
4. Head to `http://localhost:5601` in the browser
5. Click on ≡ (menu) icon in the top-left corner
6. Scroll to the bottom and click on Stack Management under the Management Section
7. In Stack Management, click on Saved Objects under Kibana in the left pane
8. Click import and then import the `kibana_dashboard.ndjson` file present in the viz folder of the github repo
9. Refresh the visualization to see real time updates.

3.4 CI/CD

We've used Jenkins for continuous integration and continuous deployment. On each push to any of the github branches, a webhook triggers the pipeline, which checks out the recent commits, builds the updated images for the app and vpn, and pushes them to a Docker Hub registry. It then runs the ansible playbook locally.

3.5 Infrastructure

We've used Ansible for configuration management and infrastructure automation. When triggered via the Jenkins pipeline, ansible first stops the existing minikube cluster and starts a fresh one with specified memory and CPU requirements. It then pulls the required images from Docker Hub, and applies all k8s manifests following which the application is deployed. Since minikube with the docker driver does not expose ports externally, we need to do port forwarding, which is also done via the playbook. Finally, the application entrypoint (Nginx→VPN), Grafana for checking metrics, and Kibana for checking logs, are available on separate ports.

3.5.1 Roles in Ansible

Ansible roles are useful to make the playbook smaller and easier to read. They split tasks into independent logical units which improves modularity, reusability and scalability. Since we had 3 namespaces `app`, `monitoring` and `logging`, we decided to have 4 roles, one for each plus one extra which creates these namespaces. This helped us to group deployments by their purpose, simplified the playbook, and will be easier to maintain in future when more functionalities might get added.

3.5.2 Ansible Vault

Ansible vault allows us to encrypt sensitive data such as passwords, API keys or other credentials within the project. We had a list of username-password pairs which were used to connect to the VPN. These were originally stored in a plain JSON file within the project, which is highly insecure. We then moved it to `secrets.yaml` in k8s which injected them at runtime, however they were still stored in plaintext. Ansible vault allows us to encrypt these secrets securely, and while running the playbook, a password or a password file can be provided to decrypt these secrets back. This is done using a Jinja2 template which renders the k8s secret from the vault file, and during playbook execution, ansible injects decrypted credentials from the vault file into the template which is then applied to k8s.

We can encrypt the secrets as:

```
ansible-vault encrypt ansible/group_vars/all/vpn_creds.yml --vault-password-
    file password.txt
```

and then we can put:

```
ansible-playbook -i inventory.ini playbook.yml --vault-password-file
    password.txt
```

in the Jenkinsfile.

4 Innovation

4.1 White-Box Application Telemetry

Most standard Kubernetes setups only monitor infrastructure metrics like CPU and RAM. We innovated by implementing **White-Box Application Telemetry** directly within our VPN's Python code using the Prometheus client library.

We instrumented our application to expose internal logic states, specifically creating custom metrics for successful VPN tunnels and invalid authentication attempts. By visualizing these in Grafana, we transformed our dashboard into a security tool. A spike in the `vpn_jwt_invalid_total` metric alerts us immediately to potential security incidents—such as token tampering or brute-force attacks that would otherwise be invisible to standard infrastructure monitoring tools.

5 Steps to Run

Ensure you have the following things installed:

- Docker: Container runtime
- Minikube & kubectl: For container orchestration
- Ansible: Configuration management
- Jenkins: For CI/CD.

If Jenkins is configured correctly with github webhooks and dockerhub creds, a push to this repository will trigger the pipeline which will start the application locally. Make sure to install the `docker-pipeline` plugin in Jenkins. Jenkins kills the port-forwarding process as soon as the job finishes. Jenkins has a feature called "Process Tree Killer" that cleans up any background processes started by a build to prevent memory leaks. Hence you will have to manually forward the ports as is in the ansible playbook.

If not, you can use Ansible:

1. Clone the repository
2. Check `ansible/playbook.yaml` and set your minikube CPU and memory specs, and other parameters
3. In the project root, run:

   ```
   ansible-playbook -i ./ansible/inventory.ini ./ansible/playbook.yml --
      vault-password-file ./ansible/password.txt

   ```

   which will start the playbook. Subsequently, services will be available on the ports specified in the playbook.
4. The main app is intuitive to use, you can log in, check a request to `/status`, and send requests at a certain rate.
5. Grafana will then show you visualizations of metrics scraped by Prometheus.
6. First create a Data view with the pattern `filebeat-*`. Then, head to the Discover tab and filter logs as per requirements (e.g. `kubernetes.labels.app:app`). For a dashboard, detailed steps are mentioned in the logging section.

6 Authors

- Prateek Rath (IMT2022017)
- Mohit Naik (IMT2022076)
- Ananthakrishna K (IMT2022086)