VIETNAM NATIONAL UNIVERSITY HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

ADVANCED SOFTWARE ENGINEERING ASSIGNMENT

DEPLOY A DEPENDABLE SYSTEM - VAULT

Major: Computer Science

Supervisors: Lê Đình Thuận

—o0o—

Students: Trần Nguyễn Phương Thành -
Đỗ Nguyễn An Huy - 2110193

HCMC, 05/2025

Contents

Chapter I: Introduction 5
1.1 About HashiCorp Vault 5
1.2 About the technology stack 5
Chapter II: Technology 6
2.1 Vault 6
2.1.1 Overview 6
2.1.2 Role in Deployment 6
2.1.3 Pros 6
2.1.4 Basic Instructions 6
2.2 ZeroTier 7
2.2.1 Overview 7
2.2.2 Role in Deployment 7
2.2.3 Pros 7
2.2.4 Basic Instructions 7
2.3 Docker Swarm 8
2.3.1 Overview 8
2.3.2 Role in Deployment 8
2.3.3 Pros 8
2.3.4 Basic Instructions 8
2.4 Prometheus 9
2.4.1 Overview 9
2.4.2 Role in Deployment 9
2.4.3 Pros 9
2.4.4 Basic Instructions 9
2.5 Grafana 10
2.5.1 Overview 10
2.5.2 Role in Deployment 10
2.5.3 Pros 10
2.5.4 Basic Instructions 10
2.6 NGINX 11
2.6.1 Overview 11
2.6.2 Role in Deployment 11
2.6.3 Pros 11
2.6.4 Basic Instructions 11
Chapter III: Design 12
Chapter IV: Implementation 13
4.1 Vault 13
4.1.1 Vault auto-unseal - Vault transit engine 13
4.1.1.1 Transit engine architecture 14
4.1.1.2 Transit server setup 14
4.1.2 Transit token authentication flow 14

4.1.2.1 Transit token generation and storage 14
4.1.2.2 Secure token access control 15
4.1.2.3 Token retrieval process 15
4.1.3 Vault cluster 16
4.1.3.1 Cluster architecture 16
4.1.3.2 Auto-unseal process 16
4.1.3.3 Node initialization 17
4.1.3.4 Startup sequence 17
4.2 Reverse Proxy / Load Balancer (NGINX) 17
4.2.1 Upstream Block 17
4.2.2 Location Block 18
4.2.3 Health and Debug Endpoints 18
4.2.4 Docker Compose Setup 18
4.2.5 Role in the System 19
4.2.5.1 Request Flow 19
4.2.5.2 Load Balancing 19
4.2.5.3 High Availability 19
4.2.5.4 Monitoring 19
4.2.6 What NGINX Can Do in the System 19
Chapter V: Deployment 20
5.1 Architecture Overview 20
5.2 Auto Unseal with Vault Transit Engine 20
5.3 Docker Swarm Deployment 20
5.4 Deployment Script 20
5.5 Monitoring Stack 22
5.6 Networking and Security 22
References 23

ID	Name	Personal work
2110193	Đỗ Nguyễn An Huy	• Setup Vault cluster with Docker Swarm • Setup Vault auto-unsealing using transit engine with Docker Swarm • Setup VPN ZeroTier
2110541	Trần Nguyễn Phương Thành	• Setup Vault cluster with Docker Swarm. • Monitor Vault Node performance with Prometheus and Grafana.

Chapter I: Introduction

In this assignment, we deploy HashiCorp Vault service (https://www.vaultproject.io/) to a virtual cluster on top of VPN with a focus on high availability.

1.1 About HashiCorp Vault

In today's digital landscape, securing sensitive information such as API keys, passwords, certificates, and other secrets is paramount for organizations of all sizes. Traditional approaches to secrets management often fall short, resulting in hard-coded credentials, shared password files, or manual distribution methods that create significant security vulnerabilities and operational inefficiencies. More modern approaches defer these responsibilities to a specialized secret management service. HashiCorp Vault plays the role of this specialized service. HashiCorp Vault provides a centralized service for securing, storing, and tightly controlling access to secrets across distributed applications and infrastructure, addressing the critical need for robust secrets management in modern technology stacks.

1.2 About the technology stack

Our implementation leverages:

- Docker Swarm: container orchestration and service management, ensuring seamless scaling and failover capabilities.
- ZeroTier: the foundation for our virtual private network, creating a secure communication layer between distributed nodes.
- NGINX: A reverse proxy and load balancer, optimizing request distribution while enhancing security.
- Prometheus & Grafana: Comprehensive monitoring and visibility by tracking system health, performance metrics, and security events.

By combining these technologies, we demonstrate a resilient, secure, and highly available HashiCorp Vault service, through which a general automatic high-availability deployment framework is presented.

Chapter II: Technology

2.1 Vault

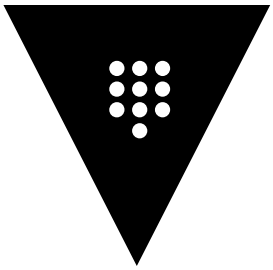

Figure 1: Vault

2.1.1 Overview

HashiCorp Vault is a secrets management tool that securely stores and controls access to tokens, passwords, certificates, API keys, and other sensitive data. It handles leasing, key revocation, key rolling, and auditing through a unified API.

2.1.2 Role in Deployment

Vault serves as the central secrets manager in our high-availability setup. It securely stores all sensitive information required by applications and infrastructure components, eliminating the need for hardcoded credentials. Its implementation addresses several critical security challenges:

- Creates a single source of truth for all secrets
- Enforces the principle of least privilege across systems
- Provides detailed audit trails for compliance requirements
- Enables automated credential rotation, reducing security risks

2.1.3 Pros

- Zero-trust security model with encryption at rest and in transit
- Dynamic secrets generation with automatic rotation
- Fine-grained access control with policy-based permissions
- Multiple authentication methods (LDAP, JWT, AWS IAM, etc.)
- Built-in audit logging and revocation capabilities

2.1.4 Basic Instructions

1. Initialize Vault server with `vault operator init`
2. Unseal the Vault using the generated unseal keys
3. Authenticate to Vault using token or authentication method
4. Configure secret engines and access policies
5. Store and retrieve secrets via CLI, API, or UI

2.2 ZeroTier

Figure 2: Zerotier

2.2.1 Overview

ZeroTier is a software-defined networking solution that creates secure, virtualized networks between devices regardless of physical location. It enables direct peer-to-peer connections through NAT traversal techniques.

2.2.2 Role in Deployment

ZeroTier establishes the secure networking layer for our Vault cluster. It creates a private, encrypted virtual network that spans across potentially disparate infrastructure, allowing:

- Secure communication between Vault nodes regardless of physical location
- Network isolation for the entire Vault deployment
- Simplified network topology without complex VPN configurations
- Consistent addressing scheme across different environments
- Reduced attack surface by eliminating public-facing services

2.2.3 Pros

- Software-defined networking without physical infrastructure changes
- End-to-end encryption for all traffic
- Works across different networks and NATs
- Centralized management of network access
- Low overhead and minimal configuration

2.2.4 Basic Instructions

1. Create a ZeroTier network in the central controller
2. Install ZeroTier client on each node
3. Join nodes to the network using network ID
4. Configure network permissions and routing
5. Verify connectivity between nodes

2.3 Docker Swarm

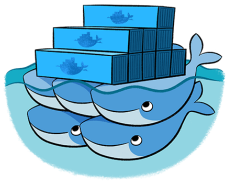

Figure 3: Docker swarm

2.3.1 Overview

Docker Swarm is a container orchestration tool built into Docker that allows you to create and manage a cluster of Docker nodes. It provides native clustering and scheduling capabilities for Docker containers.

2.3.2 Role in Deployment

Docker Swarm orchestrates our containerized Vault deployment, providing:

- Automated service placement across the cluster for high availability
- Self-healing capabilities by restarting failed containers
- Service discovery for inter-component communication
- Simplified scaling operations as demand increases
- Consistent deployment model across different environments
- Resource allocation and constraints enforcement

2.3.3 Pros

- Integrated with Docker, requiring minimal additional tools
- Simple setup and management compared to Kubernetes
- Built-in load balancing and service discovery
- Rolling updates and health checks
- Secure by default with automatic TLS encryption

2.3.4 Basic Instructions

1. Initialize a swarm with `docker swarm init`
2. Join additional nodes as managers or workers
3. Create overlay networks for service communication
4. Deploy services using Docker Compose or stack files
5. Scale services horizontally as needed

2.4 Prometheus

Figure 4: Prometheus

2.4.1 Overview

Prometheus is an open-source systems monitoring and alerting toolkit. It collects and stores metrics as time-series data, recording real-time information about the state of systems.

2.4.2 Role in Deployment

Prometheus monitors the health and performance of our Vault cluster, providing:

- Real-time visibility into system performance and resource utilization
- Early detection of potential issues through metric collection
- Historical data for capacity planning and trend analysis
- A foundation for automated alerting on abnormal conditions
- Performance insights to optimize the infrastructure

2.4.3 Pros

- Pull-based architecture reducing network complexity
- Powerful query language (PromQL)
- Multi-dimensional data model with flexible labeling
- Built-in alerting capabilities
- Service discovery integration

2.4.4 Basic Instructions

1. Configure Prometheus server using prometheus.yml
2. Set up targets (services to monitor)
3. Define scrape intervals and retention policies
4. Configure alerting rules if needed
5. Deploy exporters for system-specific metrics collection

2.5 Grafana

Figure 5: Grafana

2.5.1 Overview

Grafana is an open-source analytics and interactive visualization platform. It connects to various data sources and provides dashboards with panels representing metrics over time.

2.5.2 Role in Deployment

Grafana transforms the metrics collected by Prometheus into actionable insights through:

- Comprehensive dashboards showing system health at a glance
- Visual representation of key performance indicators
- Customizable views for different stakeholders (operators, developers, management)
- Historical performance visualization for capacity planning
- Correlation of metrics across different components of the infrastructure

2.5.3 Pros

- Rich visualization options and dashboard templates
- Multi-data source support
- Shareable, reusable dashboards
- Alerting capabilities with multiple notification channels
- Extensible with plugins and data source integrations

2.5.4 Basic Instructions

1. Install and configure Grafana server
2. Add Prometheus as a data source
3. Create or import dashboards
4. Configure panels to display relevant metrics
5. Set up alerting based on visualization thresholds

2.6 NGINX

Figure 6: NGINX

2.6.1 Overview

NGINX is a high-performance web server, reverse proxy, and load balancer. It efficiently handles connections, providing robust HTTP/HTTPS services with minimal resource usage.

2.6.2 Role in Deployment

NGINX serves as the entry point and load balancer for our Vault deployment:

- Distributes client requests across multiple Vault instances
- Provides SSL/TLS termination to offload encryption overhead
- Acts as a security barrier between external networks and Vault
- Enables seamless scaling by abstracting backend changes
- Implements circuit breaking and health checks to enhance reliability
- Provides a consistent entry point regardless of backend changes

2.6.3 Pros

- High performance with low resource utilization
- Advanced load balancing algorithms
- SSL/TLS termination and optimization
- Caching capabilities for improved response times
- Extensive configuration options and flexibility

2.6.4 Basic Instructions

1. Install NGINX on gateway nodes
2. Configure virtual hosts and proxy settings
3. Define upstream server pools for load balancing
4. Implement rate limiting and access controls

Chapter III: Design

Chapter IV: Implementation

4.1 Vault

The implementation organizes each Vault instance in its own directory:

```
services/
├── vault-1/
│   ├── .env              = Environment variables (credentials)
│   ├── .env.example      = Template for environment variables
│   ├── config.hcl        = Vault server configuration
│   ├── Dockerfile        = Container build instructions
│   ├── entrypoint.sh     = Container startup script
│   └── init.sh           = Initialization script (first node only)
├── vault-2/
│   ├── .env
│   ├── .env.example
│   ├── config.hcl
│   ├── Dockerfile
│   └── entrypoint.sh
├── vault-3/
│   ├── .env
│   ├── .env.example
│   ├── config.hcl
│   ├── Dockerfile
│   └── entrypoint.sh
├── vault-transit-1/
│   ├── .env
│   ├── .env.example
│   ├── config.hcl
│   ├── Dockerfile
│   ├── entrypoint.sh
│   └── init.sh
└── docker-compose.yml    = Service orchestration
```

`vault-transit-1/config.hcl` defines the transit server configuration:

- Uses file storage backend
- Configures API and cluster addresses

`vault-{1,2,3}/config.hcl` defines each cluster node's configuration:

- Uses Raft storage backend
- Specifies cluster addresses
- Configures the transit auto-unseal

4.1.1 Vault auto-unseal - Vault transit engine

For Vault auto-unseal, we use the simplest approach that Vault supports: the Vault Transit Engine. This implementation follows a "transit as a service" pattern, where one dedicated

Vault instance (`vault-transit-1`) provides encryption services used to unseal the other Vault instances in the cluster.

4.1.1.1 Transit engine architecture

The implementation consists of:

1. **Transit Server**: A dedicated Vault instance (`vault-transit-1`) that runs the Transit secrets engine
2. **Vault Cluster**: Three Vault nodes (`vault-1`, `vault-2`, `vault-3`) configured to use the Transit engine for auto-unsealing

This approach provides several advantages:

- Eliminates the need for manual unseal operations
- Avoids dependency on external cloud KMS services
- Maintains security by separating the transit encryption from the main Vault cluster
- Simplifies the deployment pipeline

4.1.1.2 Transit server setup

The transit server is initialized first with a single unseal key for simplicity. It:

1. Enables the Transit secrets engine
2. Creates a dedicated `autounseal` encryption key
3. Defines a policy with limited permissions for the Vault cluster nodes to use
4. Generates a token bound to this policy
5. Stores this token in a KV store for secure retrieval by cluster nodes

The initialization script creates two important policies:

- `autounseal`: Limited to encrypt/decrypt operations using the transit key
- `unseal-key`: Limited to reading the transit token from the KV store

4.1.2 Transit token authentication flow

In our Vault deployment, the auto-unseal mechanism relies on a secure token exchange between the transit server and the Vault cluster nodes. Here's how this process works:

4.1.2.1 Transit token generation and storage

The transit Vault server (`vault-transit-1`) generates a specialized transit token during initialization:

1. First, the transit server enables the transit secrets engine for encryption operations
2. It creates a dedicated encryption key called "autounseal"
3. It defines a restricted policy called "autounseal" that only allows encrypt/decrypt operations
4. It generates a token bound to this policy using `vault token create -policy=autounseal -period=24h`
5. It stores this token in a KV store at the path `kv/auto-unseal/transit-token`

This transit token is what the Vault cluster nodes need to perform auto-unseal operations. However, the token shouldn't be directly accessible without authentication.

4.1.2.2 Secure token access control

To control access to the transit token, the transit server:

1. Creates a policy called "unseal-key" with very limited permissions:

```
path "kv/auto-unseal/transit-token" {
  capabilities = ["read"]
}
```

   This policy only permits reading the transit token and nothing else.

2. Sets up userpass authentication:

```
vault auth enable userpass
vault write auth/userpass/users/internal-server
  password=$UNSEAL_PASSWORD
  policies=unseal-key
```

   This creates a service account that can only access the transit token.

The password is stored as an environment variable in the `.env` file mounted to each Vault container, keeping it secure and configurable.

4.1.2.3 Token retrieval process

When a Vault cluster node starts up, it follows this authentication flow to retrieve the transit token:

1. Authenticate to the transit server using the userpass credentials:

```
VAULT_TOKEN=$(curl -s
    --request POST
    --data "{\"password\": \"$UNSEAL_PASSWORD\"}"
    http://vault-transit-1:8200/v1/auth/userpass/login/internal-
server | jq -r '.auth.client_token')
```

2. Validate the authentication succeeded:

```
if [ "$VAULT_TOKEN" == "null" ] || [ -z "$VAULT_TOKEN" ]; then
    echo "Authentication failed"
    exit 1
fi
```

3. Use the authenticated token to retrieve the transit token:

```
TRANSIT_TOKEN=$(curl -s
    --header "X-Vault-Token: $VAULT_TOKEN"
    http://vault-transit-1:8200/v1/kv/auto-unseal/transit-token |
jq -r '.data.token')
```

4. Validate the transit token was successfully retrieved:

```
if [ "$TRANSIT_TOKEN" == "null" ] || [ -z "$TRANSIT_TOKEN" ]; then
    echo "Failed to retrieve transit token"
    exit 1
fi
```

5. Set the transit token as the environment variable for Vault:

```
export VAULT_TOKEN=$TRANSIT_TOKEN
```

This multi-step authentication process ensures that only authorized Vault nodes can access the transit token needed for auto-unsealing operations. The temporary authentication token from userpass is only used to retrieve the actual transit token, which is then used for the auto-unseal process.

This security architecture creates a chain of trust where the transit server manages access to the encryption key, while still allowing automated unsealing without human intervention.

4.1.3 Vault cluster

Each Vault service in the cluster uses the Raft storage engine for data persistence and replication. Raft provides:

- Built-in high availability without external dependencies
- Strong consistency across the cluster
- Automatic leader election and failover
- Simple setup compared to external storage options

4.1.3.1 Cluster architecture

The Vault cluster follows a leader-follower topology:

- `vault-1` initializes as the first node and becomes the leader
- `vault-2` and `vault-3` join the cluster as followers
- All nodes can serve read requests
- Only the leader processes write operations

4.1.3.2 Auto-unseal process

The auto-unseal flow works as follows:

1. Each Vault node starts up and authenticates to the transit server

2. The node retrieves the transit token needed for auto-unseal operations
3. The node configures its seal type to use the transit server
4. When sealed, the node sends its encryption key to the transit server for decryption
5. The transit server decrypts and returns the key, allowing the node to unseal

4.1.3.3 Node initialization

Vault-1 has special handling as the first node:
1. Waits for the transit server to be available
2. Retrieves the transit token
3. Initializes itself if not already initialized
4. Becomes the leader of the Raft cluster

Vault-2 and Vault-3 follow a similar but simpler process:
1. Wait for vault-1 to be available
2. Retrieve the transit token
3. Join the Raft cluster using the `raft join` command
4. Start serving requests once joined

4.1.3.4 Startup sequence

The deployment follows a carefully orchestrated startup sequence:

1. `vault-transit-1` initializes first and becomes available for auto-unseal operations
2. `vault-1` waits for the transit server, then initializes as the first node in the cluster
3. `vault-2` and `vault-3` wait for `vault-1` to be ready, then join the cluster

This sequence ensures that dependencies are satisfied before each service attempts to start, preventing race conditions and initialization failures.

4.2 Reverse Proxy / Load Balancer (NGINX)

NGINX's configuration in this project is tailored to support the Vault cluster. Here's how it's set up:

4.2.1 Upstream Block

The `upstream` directive defines the group of backend Vault servers:

```
upstream vault_backend {
    server vault-1:8200;
    server vault-2:8200;
    server vault-3:8200;
    keepalive 64;
}
```

- **Servers**: Lists three Vault nodes (`vault-1`, `vault-2`, `vault-3`) on port 8200.
- **Keepalive**: Maintains 64 persistent connections per server, reducing connection overhead.

4.2.2 Location Block

The `location` block handles requests to the root path (`/`):

```
location / {
    proxy_pass http://vault_backend;
    proxy_set_header Host $host;
    proxy_set_header X-Real-IP $remote_addr;
    proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
    proxy_set_header X-Forwarded-Proto $scheme;
    proxy_intercept_errors on;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_connect_timeout 30s;
    proxy_send_timeout 60s;
    proxy_read_timeout 60s;
}
```

- **`proxy_pass`**: Forwards requests to the `vault_backend` group, enabling load balancing.
- **Headers**: Passes client details (e.g., IP, protocol) to Vault nodes for logging and security.
- **Timeouts**: Sets limits for connection (30s), sending (60s), and reading (60s) operations.
- **HTTP/1.1**: Supports persistent connections and WebSocket upgrades.

4.2.3 Health and Debug Endpoints

Additional endpoints provide monitoring and troubleshooting:

```
location /health {
    return 200 "NGINX is healthy\n";
}

location /debug {
    return 200 "Debug information: $time_local\n";
}
```

- **`/health`**: Returns a status message for monitoring tools.
- **`/debug`**: Provides the current server time for diagnostics.

4.2.4 Docker Compose Setup

NGINX is deployed via Docker Compose:

```
nginx:
  image: nginx
```

```
  hostname: nginx
  ports:
    - "80:80"
  networks:
    - vault-network
  deploy:
    mode: replicated
    replicas: 3
```

- **Replicas**: Runs three NGINX instances for redundancy.
- **Ports**: Maps port 80 on the host to NGINX.
- **Network**: Connects to `vault-network` for Vault communication.

4.2.5 Role in the System

NGINX integrates seamlessly with the Vault cluster, enhancing its functionality:

4.2.5.1 Request Flow

1. A client sends a request to NGINX (e.g., `http://nginx:80/`).
2. NGINX selects a Vault node from `vault_backend` and forwards the request.
3. The Vault node processes it and responds to NGINX.
4. NGINX relays the response to the client.

4.2.5.2 Load Balancing

- Distributes requests evenly across `vault-1`, `vault-2`, and `vault-3`.
- Ensures optimal resource use and prevents bottlenecks.

4.2.5.3 High Availability

- Three NGINX replicas provide failover; if one fails, traffic shifts to others.
- NGINX skips unavailable Vault nodes, maintaining service continuity.

4.2.5.4 Monitoring

- The `/health` endpoint allows external tools to verify NGINX's status.
- The `/debug` endpoint aids in troubleshooting.

4.2.6 What NGINX Can Do in the System

NGINX's capabilities elevate the Vault cluster:

- **Scalability**: Easily add more Vault nodes to `vault_backend` as traffic grows.
- **Performance**: Reduces latency with persistent connections and efficient load distribution.
- **Resilience**: Ensures uptime with multiple NGINX instances and failover.
- **Security**: Hides Vault nodes and can be extended with HTTPS (currently missing).

Chapter V: Deployment

We use Docker Swarm [1] to deploy our Vault-based Architecture. The system consists of multiple Vault nodes (including one for transit auto-unseal), NGINX as a reverse proxy, and Prometheus and Grafana for monitoring. The deployment uses custom Vault images and automated configuration through a shell script.

5.1 Architecture Overview

The deployment includes the following services:

- `vault-1`, `vault-2`, `vault-3`: A three-node Vault cluster forming a High Availability (HA) setup.
- `vault-transit-1`: A dedicated Vault instance running the Transit secrets engine for auto-unsealing other Vault nodes.
- `nginx`: Acts as a reverse proxy for routing traffic to Vault and monitoring endpoints.
- `prometheus`: Collects metrics from services for monitoring.
- `grafana`: Visualizes metrics collected by Prometheus.

All services are connected through a Docker Swarm overlay network named `vault-network`.

5.2 Auto Unseal with Vault Transit Engine

To enable auto unseal, the primary Vault nodes (`vault-1`, `vault-2`, `vault-3`) are configured to use the Transit secrets engine hosted on `vault-transit-1`. This method avoids manual re-unsealing on restarts and improves automation and resilience.

5.3 Docker Swarm Deployment

All services are deployed in Docker Swarm using a `docker-compose.yml` file. Key configurations include:

- Each Vault node runs in a single replica mode for strict control.
- Prometheus, Grafana, and NGINX run in replicated mode (3 replicas each) to support load balancing and high availability.
- Persistent volumes are defined for all Vault nodes and Grafana to ensure data durability.
- The Prometheus configuration is injected using Docker configs.

5.4 Deployment Script

```
#!/bin/sh
set -e

echo "Cleaning up old Docker services, containers, volumes,
images..."
```

```
docker service rm vault_vault-1 vault_vault-2 vault_vault-3
vault_vault-transit-1 vault_prometheus vault_grafana || echo
"Services not created yet"

for IMAGE in vault-1 vault-2 vault-3 vault-transit-1; do
  docker rm $(docker ps -a --filter ancestor=$IMAGE --
format="{{.ID}}") --force 2>/dev/null || echo "Containers for $IMAGE
not found"
done

docker volume rm vault_vault-1-data vault_vault-2-data vault_vault-3-
data vault_vault-transit-1-data grafana-data || echo "Volumes not
created yet"

docker rmi vault-1 vault-2 vault-3 vault-transit-1 --force || echo
"Images not created yet"

echo "Cleaning old Docker config for Prometheus (if exists)..."
docker config rm prometheus-config 2>/dev/null || echo "No existing
prometheus-config to remove"

echo "Creating new Docker config for Prometheus..."
docker config create prometheus-config ./prometheus/prometheus.yml

echo "Initializing Docker Swarm (if not active)..."
if ! docker info --format '{{.Swarm.LocalNodeState}}' | grep -q
"active"; then
  docker swarm init || echo "Swarm already initialized"
fi

echo "🔨 Building Vault service images..."
for SERVICE in vault-1 vault-2 vault-3 vault-transit-1; do
  echo "Building $SERVICE..."
  (cd $SERVICE && docker build -t $SERVICE .)
done

echo "Deploying full Vault stack with Prometheus and Grafana..."
docker stack deploy --compose-file=docker-compose.yml vault

echo "Vault stack deployment complete."
```

The deployment is managed by a shell script that performs the following tasks:

1. Cleans up old containers, volumes, and images related to Vault and monitoring stack.
2. Deletes and recreates the Prometheus Docker config.
3. Initializes Docker Swarm if it hasn't been already.
4. Builds custom Docker images for all Vault-related services.
5. Deploys the entire stack using `docker stack deploy`.

5.5 Monitoring Stack

- **Prometheus**: Scrapes metrics from services. Configuration is provided via an external config file.
- **Grafana**: Provides dashboards for visualizing metrics. Persists data using a named volume (`grafana-data`).

5.6 Networking and Security

- All services share an `overlay` network (`vault-network`) which is attachable, facilitating secure inter-service communication.
- Vault containers are granted `IPC_LOCK` capability for memory locking, improving security.

References

[1] TestDriven.io, “Deploying Vault and Consul with Docker Compose.”