GEORG-AUGUST-UNIVERSITÄT GÖTTINGEN IN PUBLICA COMMODA SEIT 1737

HPS

Seminar Report

Kubernetes for HPC

Pranay Bhatia

MatrNr: 17935037

Supervisor: Jonathan Decker

Georg-August-Universität Göttingen
Institute of Computer Science

March 9, 2024

Abstract

Automating Kubernetes Pod Restarts with Dynamic Environment Variable Injection using HashiCorp Vault

In the realm of Kubernetes orchestration, automating the dynamic injection of environment variables into running pods has been a longstanding challenge. This research introduces a novel solution that leverages HashiCorp Vault and a Kubernetes liveness probe to facilitate automated restarts of pods upon changes to environment variables.

Traditional solutions for injecting secrets into pods at startup lack the capability to handle dynamic updates without manual intervention. This research addresses the difficulty of achieving automated restarts when environment variables within a pod need to be modified, presenting a comprehensive solution for seamless integration.

Current industry-standard solutions, such as HashiCorp Vault, excel at secret injection during pod initialization. However, they fall short in automating pod restarts when environment variables change, requiring manual restarts or reliance on DevOps processes. This limitation hinders the efficiency and agility of managing dynamic configurations.

Our innovative approach introduces a liveness probe within Kubernetes, ensuring constant monitoring of the environment variable path within the pod. If changes are detected, the liveness probe communicates with HashiCorp Vault, retrieves the updated variables, and triggers a controlled restart of the main container. This self-contained solution eliminates the need for manual interventions and provides a seamless mechanism for automated updates.

The proposed solution has been successfully implemented, demonstrating its effectiveness in automating pod restarts upon changes to environment variables. The evaluation revealed that the solution operates with minimal overhead, ensuring a reliable and efficient mechanism for maintaining up-to-date configurations within a dynamic Kubernetes environment.

In summary, this research introduces an automated solution to a prevalent challenge in Kubernetes orchestration, offering a streamlined process for dynamically updating environment variables and triggering pod restarts. The outcome demonstrates the potential for increased efficiency and autonomy in managing containerized applications, paving the way for more responsive and adaptive Kubernetes deployments.

Statement on the usage of ChatGPT and similar tools in the context of examinations

In this work I have used ChatGPT or a similar AI-system as follows:

- ☐ Not at all
- ☐ In brainstorming
- ☐ In the creation of the outline
- ☑ To create individual passages, altogether to the extent of 60% of the whole text
- ☐ For proofreading
- ☐ Other, namely: -

I assure that I have stated all uses in full.
Missing or incorrect information will be considered as an attempt to cheat.

Contents

List of Tables

List of Figures

List of Listings

List of Abbreviations

HPC High-Performance Computing

1 Introduction

In modern containerized environments orchestrated by Kubernetes, managing environment variables efficiently and securely is essential for ensuring the reliability and security of applications. Environment variables play a crucial role in configuring and customizing containerized applications, providing a flexible and dynamic way to pass configuration information to running containers. However, manually updating environment variables in Kubernetes deployments can be cumbersome and error-prone, particularly in dynamic environments where configuration changes are frequent.

The aim of this report is to propose a novel approach for smartly injecting environment variables into Kubernetes clusters and automating the process of detecting changes and restarting pods accordingly. By leveraging Kubernetes' liveness probe mechanism—a built-in feature for determining the health of containerized applications—we can create a robust and automated solution for managing environment variables seamlessly.

Our approach involves encapsulating the logic for retrieving, validating, and updating environment variables within a liveness probe script, which is embedded within the container's configuration. This script continuously monitors the availability and integrity of environment variables, fetching them from a secure and centralized source, such as HashiCorp Vault. Upon detecting changes or discrepancies in the environment variables, the liveness probe triggers a pod restart, ensuring that the application remains up-to-date with the latest configuration changes.

By automating the injection and updating of environment variables in Kubernetes deployments, our approach offers several benefits, including:

- Improved Reliability: Ensuring that applications always have access to the correct and up-to-date environment variables enhances the reliability and stability of Kubernetes deployments.
- Enhanced Security: By fetching environment variables from a centralized and secure source, such as HashiCorp Vault, we can enforce access controls, encryption, and audit trails, thereby enhancing the security posture of the application.
- Efficient Configuration Management: Automating the process of injecting and updating environment variables simplifies configuration management tasks, reducing the risk of human error and streamlining the deployment process.
- Dynamic Scalability: With automated environment variable injection and pod restarts, Kubernetes deployments can dynamically scale in response to changing workload demands while ensuring consistent and reliable configuration across all instances.

In this report, we present the methodology, implementation details, experimental results, and discussion of our approach for smartly injecting environment variables in Kubernetes clusters. Through empirical evaluation and analysis, we demonstrate the effectiveness and efficiency of our solution in improving the reliability, security, and manageability of Kubernetes deployments.

2 Problem Statement

In Kubernetes deployments, managing environment variables effectively while ensuring their timely and accurate injection into running pods poses a significant challenge. Manually updating environment variables can lead to errors, inconsistencies, and downtime, particularly in dynamic environments where configuration changes are frequent. The problem statement revolves around the need for a robust and automated solution to intelligently inject environment variables into Kubernetes clusters and automatically trigger pod restarts upon detecting changes or updates to the configuration.

Key challenges include:

1. **Manual Configuration Management:** Manually updating environment variables in Kubernetes deployments is time-consuming, error-prone, and can result in configuration drifts and inconsistencies across pods.

2. **Dynamic Environment:** In dynamic Kubernetes environments where pods scale up, scale down, or migrate across nodes, ensuring that all pods have access to the latest environment variables becomes challenging.

3. **Security and Compliance:** Ensuring the security and confidentiality of environment variables while maintaining compliance with regulatory requirements, such as GDPR and HIPAA, is critical for protecting sensitive information.

4. **Application Reliability:** Changes to environment variables, such as API keys or database credentials, may require pod restarts to take effect, impacting application availability and reliability if not handled efficiently.

5. **Scalability:** As Kubernetes deployments scale to accommodate varying workloads, the process of injecting and updating environment variables must scale accordingly to meet the demands of dynamic environments.

Addressing these challenges requires the development of an automated solution that integrates seamlessly with Kubernetes deployments, intelligently manages environment variables, and ensures the reliability, security, and scalability of applications running in Kubernetes clusters.

3 Background and Motivation

3.1 HashiCorp Vault

Introduction: HashiCorp Vault is a popular open-source tool for managing secrets and sensitive data in modern cloud-native environments. It provides a centralized platform for securely storing, accessing, and managing secrets such as API keys, passwords, certificates, and encryption keys. Vault offers a robust set of features and capabilities designed to address the challenges of secrets management in dynamic and distributed systems.

Use and Benefits: The primary use of HashiCorp Vault is to securely manage secrets and sensitive data in cloud-native applications and infrastructure. By centralizing the

storage of secrets, Vault helps organizations improve security, compliance, and operational efficiency. Some key benefits of using HashiCorp Vault include:

- **Enhanced Security:** Vault provides encryption, access control, and auditing capabilities to protect sensitive data from unauthorized access and breaches.
- **Scalability:** Vault is designed to scale horizontally to meet the needs of large and dynamic environments, ensuring that secrets management remains efficient and reliable as deployments grow.
- **Compliance:** Vault helps organizations meet regulatory compliance requirements by enforcing security best practices and providing audit trails for secret access and management.
- **Integration:** Vault seamlessly integrates with popular cloud platforms, container orchestration systems (such as Kubernetes), and CI/CD pipelines, making it easy to incorporate secrets management into existing workflows.

Thanks to Built-in APIs: One of the key features of HashiCorp Vault is its rich set of built-in APIs, which allow developers and operators to interact with Vault programmatically. These APIs provide access to Vault's functionality, including the ability to retrieve, create, update, and delete secrets. Additionally, Vault offers APIs for dynamic secrets generation, token management, and encryption operations. This flexibility enables developers to automate secrets management tasks and integrate Vault seamlessly into their applications and infrastructure.

Available APIs to Fetch Secrets: Vault exposes several APIs for fetching secrets, each designed to accommodate different use cases and access patterns:

- **Key/Value Secrets Engine API:** This API allows users to store and retrieve secrets as key/value pairs within Vault's hierarchical data store.
- **Database Secrets Engine API:** Vault provides APIs for dynamically generating and managing database credentials, allowing applications to securely access databases without exposing static credentials.
- **AWS Secrets Engine API:** With this API, Vault can dynamically generate AWS IAM credentials with configurable TTLs and IAM policies, enabling secure access to AWS resources.
- **Kubernetes Secrets Engine API:** Vault integrates seamlessly with Kubernetes clusters, allowing applications running in Kubernetes to authenticate with Vault and fetch secrets securely using Kubernetes service accounts.

In summary, HashiCorp Vault is a powerful tool for managing secrets and sensitive data in cloud-native environments. Its rich set of features, flexible APIs, and seamless integrations make it an essential component of modern infrastructure and application security architectures.

3.2 Liveness Probes

Functionality of Liveness Probe

What is Liveness Probe: In Kubernetes, a liveness probe is a mechanism used to determine the health of a containerized application running in a pod. It periodically checks the application's state to ensure that it is running as expected. The liveness probe performs this check by sending requests to a specified endpoint within the container and evaluating the response. If the application responds successfully, indicating that it is healthy, the liveness probe considers the container to be in a good state. However, if the application fails to respond or returns an error, indicating that it is unhealthy, the liveness probe takes action, such as restarting the container.

Benefits: The use of liveness probes offers several benefits for managing containerized applications in Kubernetes deployments:

- **Improved Reliability:** By continuously monitoring the health of containerized applications, liveness probes help ensure that unhealthy containers are detected and replaced promptly, minimizing downtime and enhancing application reliability.
- **Automatic Recovery:** Liveness probes facilitate automatic recovery from application failures by restarting unhealthy containers, thereby maintaining the desired state of the application and preserving service availability.
- **Scalability:** Liveness probes enable Kubernetes clusters to dynamically scale resources based on workload demands, as unhealthy containers are replaced with healthy ones, allowing the cluster to adapt to changing conditions efficiently.
- **Simplified Operations:** By automating the detection and recovery of unhealthy containers, liveness probes reduce the need for manual intervention and streamline operations, making it easier to manage large-scale Kubernetes deployments.

Parameters: Liveness probes in Kubernetes are configured using several parameters that define how the probe operates and when it should take action:

- **Initial Delay:** This parameter specifies the amount of time to wait after the container starts before performing the first liveness probe. It allows the application to initialize before being checked for health.
- **Period:** The period parameter defines the frequency at which the liveness probe should be executed, indicating how often the application's health should be assessed.
- **Timeout:** This parameter sets the maximum amount of time the probe should wait for a response from the application. If the application fails to respond within this timeframe, the probe considers it unhealthy.
- **Success Threshold:** The success threshold parameter specifies the number of consecutive successful probe results required to consider the application as healthy.
- **Failure Threshold:** Conversely, the failure threshold parameter defines the number of consecutive failed probe results that indicate the application is unhealthy and should be restarted.

Leveraging Liveness Probe for Conditional Restart of Cluster: By leveraging the functionality of liveness probes, Kubernetes deployments can implement conditional restarts of clusters based on the health status of containerized applications. This approach involves configuring liveness probes to monitor critical components or services within the cluster and trigger restarts if anomalies or failures are detected. For example, if a key service within the cluster becomes unresponsive or experiences errors, the corresponding liveness probe can detect this condition and initiate a restart of the affected containers or pods. This proactive approach to managing cluster health helps maintain overall system reliability and availability, ensuring that applications continue to operate smoothly even in the face of transient failures or issues.

4 Experimental Setup

4.1 Environment Configuration

In the experimental setup, the environment configuration entails the setup of both the Kubernetes cluster and the HashiCorp Vault instance.

For the Kubernetes cluster, a standard setup using a cloud provider or a local development environment, such as Minikube or Kind, can be used. The cluster should be configured with sufficient resources to deploy the application pods and accommodate any additional components required for integration with Vault.

The HashiCorp Vault instance should be deployed and configured to serve as the central secrets management platform. This includes configuring authentication methods, such as tokens or Kubernetes service accounts, enabling the necessary secrets engines (e.g., Key/Value, AWS, Database), and defining access policies to control who can access which secrets.

Additionally, environment variables should be set up within the Kubernetes cluster to provide necessary configuration parameters to the script, such as `VAULT_ENABLE`, `VAULT_TOKEN`, `VAULT_SERVICE_HOST`, and `VAULT_SERVICE_PORT`. These variables should be securely managed and protected to prevent unauthorized access.

4.2 Implementation Details

The implementation details section provides insights into how the script provided in the previous section is integrated into the Kubernetes deployment and how it interacts with HashiCorp Vault.

Firstly, the script should be packaged along with the application container image or as a ConfigMap in Kubernetes. This ensures that it is available within the container environment and can be executed as part of the pod lifecycle.

During pod initialization, Kubernetes mounts the script and executes it as a liveness probe. The script then communicates with HashiCorp Vault using the specified address and token to fetch secrets as needed.

The script utilizes standard Unix commands, such as `curl` and `jq`, to interact with the Vault API and parse the JSON responses. It also handles various HTTP status codes returned by the API calls to handle different scenarios, such as initializing a new secret store or updating environment variables.

Upon detecting changes in the retrieved secrets, the script exports them as environment variables within the pod, ensuring that the application has access to the latest configuration. If changes are detected, the script exits with a non-zero status code, triggering Kubernetes to restart the pod and apply the updated environment variables.

Overall, the implementation details highlight the seamless integration of the script with Kubernetes and its interaction with HashiCorp Vault to ensure the continuous availability and integrity of secrets within the containerized environment.

5 Methodology

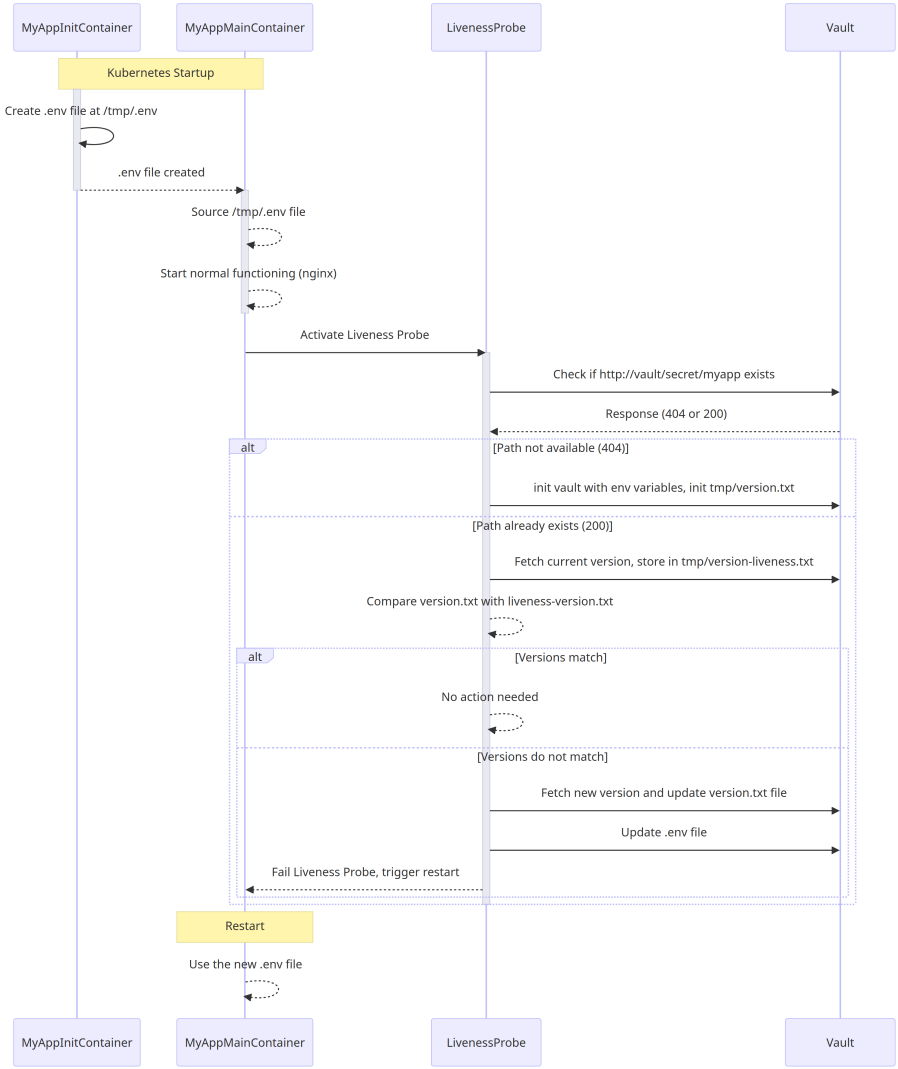

Figure 1: implementation Sequence diagram

5.1 Kubernetes and Vault Integration

The script provided facilitates the integration between Kubernetes and HashiCorp Vault. It begins by checking if the `VAULT_ENABLE` environment variable is set to "true". If

enabled, it proceeds to fetch secrets from Vault using the specified token and address constructed based on environment variables (`VAULT_SERVICE_HOST` and `VAULT_SERVICE_PORT`). The script then handles various HTTP status codes returned by Vault API calls, such as 404 for a missing secret store or 403 for an incorrect token. Additionally, it supports initializing a new secret store if not found, exporting retrieved secrets as environment variables, and updating them as needed.

5.2 Liveness Probe Implementation

The liveness probe implementation leverages Kubernetes' built-in feature to ensure the health of containerized applications. Within the script, an HTTP request is made to the specified Vault address to check the availability of secrets. The response status code is evaluated, and actions are taken accordingly. If the secret store is missing (404), it initializes a new store. If the token is incorrect (403), it logs an error. Otherwise, it compares the version of the retrieved secrets with the locally stored version. If a mismatch is detected, it updates the environment variables and exports them for the application to use.

5.3 Deployment Restart Mechanism

The deployment restart mechanism is triggered conditionally based on the outcome of the liveness probe. If the script detects changes in the environment variables fetched from Vault, it exits with a non-zero status code (1), indicating that a restart is required. This triggers Kubernetes to restart the pod, ensuring that the application picks up the updated environment variables. Conversely, if no changes are detected, the script exits with a zero status code (0), indicating that the environment variables are up to date, and no restart is necessary. This mechanism ensures that the application remains resilient to configuration drifts and always uses the latest secrets from Vault.

6 Conclusion

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

6.1 Citation example

Hawthorn et al. [HWS01] talk about laser, which is similar to the work of his colleagues [WH91; Arn+98].

6.2 Table and listing example

Table 1 shows an example for a table in LaTeX.

Table 1: The effects of treatments X and Y on the four groups studied.

Groups	Treatment X	Treatment Y
1	0.20	0.80
2	0.17	0.70
3	0.24	0.75
4	0.68	0.30

[If you show results in tables, always ensure all of them are aligned to the right and have the same precision.]

Listing 1 shows an example of a listing, a good way to display code in LaTeX.

```
package main
import "fmt"
func main() {
    fmt.Println("Hello, world!")
}
```

Listing 1: "Hello, world!" in Go

References

[Arn+98] A. S. Arnold et al. "A Simple Extended-Cavity Diode Laser". In: *Review of Scientific Instruments* 69.3 (Mar. 1998), pp. 1236–1239. URL: http://link.aip.org/link/?RSI/69/1236/1.

[HWS01] C. J. Hawthorn, K. P. Weber, and R. E. Scholten. "Littrow Configuration Tunable External Cavity Diode Laser with Fixed Direction Output Beam". In: *Review of Scientific Instruments* 72.12 (Dec. 2001), pp. 4477–4479. URL: http://link.aip.org/link/?RSI/72/4477/1.

[WH91] Carl E. Wieman and Leo Hollberg. "Using Diode Lasers for Atomic Physics". In: *Review of Scientific Instruments* 62.1 (Jan. 1991), pp. 1–20. URL: http://link.aip.org/link/?RSI/62/1/1.

A Code samples

```
if [ "$VAULT_ENABLE" = "true" ]; then

    VAULT_TOKEN="admin"
    VAULT_PATH="myapp"
    VAULT_ADDRESS="http://$VAULT_SERVICE_HOST:$VAULT_SERVICE_PORT/v1/secret/data/$VAULT_PATH"
    echo "address: " $VAULT_ADDRESS
    VAULT_HEADER="X-Vault-Token: $VAULT_TOKEN"

    echo "Installing necessary packages: curl and jq..."
    which apt-get && apt-get update && apt-get install -y curl jq
    curl
    jq

    HTTP_STATUS_CODE=$(curl -s --location "$VAULT_ADDRESS" --header "X-Vault-Token: $VAULT_TOKEN" -o /dev/null -w "%{http_code}")

    if [ "$HTTP_STATUS_CODE" = "000" ]; then
        echo "Service not found on given URL: $VAULT_ADDRESS"
        exit 0
    elif [ "$HTTP_STATUS_CODE" = "404" ]; then
        echo "Store does not exist. Initializing Vault store in $VAULT_PATH for the first time..."
        BODY=$(env | sort | jq -R -n 'reduce inputs as $line ({}; . + ($line | split("=") | {(.[0]): .[1]})) | {"data": .}')
        curl -s -H "$VAULT_HEADER" -H "Content-Type: application/json" -X POST -d "$BODY" "$VAULT_ADDRESS" -o /dev/null
        echo 1 > /tmp/version.txt
        exit 0
    elif [ "$HTTP_STATUS_CODE" = "403" ]; then
        echo "Incorrect token: $VAULT_TOKEN for $VAULT_ADDRESS"
        exit 0
    else
        echo "Status code: $HTTP_STATUS_CODE"
        echo "$(curl -s -H "$VAULT_HEADER" "$VAULT_ADDRESS" | jq -r '.data.metadata.version')" > /tmp/liveness-version.txt

        if cmp -s /tmp/version.txt /tmp/liveness-version.txt; then
            echo "Environment variable version $(cat /tmp/liveness-version.txt) is up to date"
            exit 0
        else
            echo "New environment variable version $(cat /tmp/liveness-version.txt) detected"
```

```
37              JSON_DATA="$(curl -s -H "$VAULT_HEADER" "$VAULT_ADDRESS
                   ↪ ")"
38              echo "$( echo $JSON_DATA | jq -r '.data.metadata.
                   ↪ version')" > /tmp/version.txt
39              rm /tmp/.env || true
40              echo "Exporting variables for $VAULT_PATH with version
                   ↪ $(cat /tmp/version.txt)"
41              keys=$(echo "$JSON_DATA" | jq -r '.data.data | keys[]')
42              for key in $keys; do
43                  value=$(echo "$JSON_DATA" | jq -r ".data.data[\"
                       ↪ $key\"]")
44                  export "$key"="$value"
45                  # echo "Exporting variable: $key=$value"
46                  echo "$key=\"$value\"" >> /tmp/.env
47              done
48              exit 1
49          fi
50      fi
51  else
52      echo "VAULT_ENABLE is not set to true. Doing nothing and moving
           ↪  on as expected."
53  fi
```

Listing 1: liveness probe bash script