Redesigning and Implementing the Bootstrap of Large Scale Kubernetes Enterprise Infrastructure through Automated Self Contained CLI

Project 2b (T3_2000)

in the context of the examination for the

Bachelor of Science (B.Sc.)

in Computer Science (Informatik)

at the Cooperative State University Baden-Württemberg Karlsruhe

by

Yannik Schiebelhut

Submission Date:	September 19, 2022
Processing period:	July 4, 2022 - September 19, 2022
Matriculation number, Course:	3354235, TINF20B1
Training institution:	SAP SE Dietmar-Hopp-Allee 16 69190 Walldorf, Deutschland
Supervisor of the training institution:	Samed Guener
Appraiser of the Cooperative State University:	Prof. Dr. Johannes Freudenmann

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Project 2b (T3_2000) mit dem Thema:

> *Redesigning and Implementing the Bootstrap of Large Scale Kubernetes Enterprise Infrastructure through Automated Self Contained CLI*

gemäß § 5 der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. September 2017 selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Karlsruhe, den 22. August 2022

Schiebelhut, Yannik

Sperrvermerk

Die nachfolgende Arbeit enthält vertrauliche Daten der:

SAP SE
Dietmar-Hopp-Allee 16
69190 Walldorf, Deutschland

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anderslautende Genehmigung vom Dualen Partner vorliegt.

Abstract

- Deutsch -

Platzhalter

Abstract

- English -

placeholder

Contents

List of Abbreviations **VI**

List of Figures **VII**

List of Code Listings **VIII**

1 Introduction **1**
1.1 Motivation 1
1.2 Goal 1
1.3 Approach 2

2 Fundamentals **3**
2.1 Go 3
2.2 Terraform 4
2.3 Kubernetes 5
2.4 Gardener 6
2.5 Amazon Web Services (AWS) 7
2.6 Jenkins 8
2.7 Vault 8

3 Conceptional Thoughts **10**
3.1 Interaction with Services 10
3.2 Assumptions 13
3.3 Policy Management 13

4 Implementation **14**
4.1 AWS Client Struct 14
4.2 Checking AWS State and Creating Objects 17
4.3 Policy Generation from File 19
4.4 Key Rotation 21
4.5 Updating Keys in Vault 24

5 Future Work **26**
5.1 Establishing Tests for the Bootstrap 26
5.2 Integration with the command-line interface (CLI) 28

6 Evaluation **30**

Literaturverzeichnis **IX**

List of Abbreviations

ACL Access Control List
API Application Programming Interface
ARN Amazon Resource Name
AWS Amazon Web Services
CICD Continuous Integration and Continuous Delivery
CLI command-line interface
EKS Elastic Kubernetes Service
HCL HashiCorp Configuration Language
HTTP Hypertext Transfer Protocol
IaaS Infrastructure as a Service
IaC Infrastructure as Code
IAM Identity and Access Management
JSON JavaScript Object Notation
REST Representational State Transfer
S3 Simple Storage Service
SAML Security Assertion Markup Language
SDK Software Development Kit
STS Security Token Service
URL Uniform Resource Locator
VCS Version Control System

List of Figures

2.1 Gardener architecture [11] 7

List of Code Listings

4.1 AWS struct (excerpt from aws.go) 15
4.2 Constructor for the AWS Struct (excerpt from aws.go) 16
4.3 Creating the Terraform State Bucket (excerpt from aws.go) 18
4.4 Template File for the AWS Policy (excerpt) 20
4.5 Method for Policy Generation (excerpt from aws.go) 20
4.6 Key Rotation (excerpt from aws.go) 22
4.7 Saving Keys to Vault 24

1 Introduction

1.1 Motivation

Modern enterprise infrastructure for software development often makes use of cloud computing to dynamically adapt infrastructure to the fast-paced world of agile development. The setup process of one so-called cluster is traditionally done by hand and can easily require multiple days of work to complete. Obviously, in consequence of the required man-hours, this manual deployment is rather costly and time-consuming. Also, humans are prune to error. In a manual task that big, it is very likely for unforeseeable errors to occur. Finding these mistakes can eventually take additional time.

As a conclusion, it is desired to automate as much of a clusters' deployment process as possible. There are already various tools available that can be leveraged to perform an automated deployment of a declared cluster infrastructure. But even these automatization tools require some perquisites to run, like a technical access user with access to the necessary resources and access keys to make use of this user, which still have to be created manually or with the help of bare-bones scripts. The process of fulfilling these requirements is referred to as bootstrap.

1.2 Goal

Goal of this project is to simplify the bootstrap of a cluster by integrating the necessary functionality into a CLI which is currently developed by the supervising department. Also, the usage should be convenient and, beside providing login data for the cloud provider and credential store, minimal user input should be required to perform the bootstrap. Furthermore, the newly developed CLI is required to be able to get executed by an automated build server. The ways of user interaction have to be designed accordingly.

1.3 Approach

First, the fundamentals of the tools and services that will be worked with are introduced. Most of these tools are fixed because of the current working set of the supervising department and because the target services that have to be interacted with are clearly defined. Although, some conceptional thoughts have to be put into the appropriate choice of libraries and Application Programming Interfaces (APIs) that are used as a bridge between the programming language and the actual remote services. As an orientation for the bootstrapping procedure, an existing Python script will be leveraged. This script is checked for possible improvements and modernization potential. The newly developed procedure is then implemented as a general proof of concept. After this, unit tests will be established to prospectively ensure the structural integrity of the developed solution. Then, the solution has to be integrated with the CLI. Eventually, the developed bootstrap will be analyzed regarding economical aspects and possible weak points and a conclusion will be drawn.

adapt this to the actual implementation chapter

2 Fundamentals

2.1 Go

Go (also known as Golang) is an open source programming language that was started at Google in 2007 and initially launched in 2009. The language was designed to face engineering challenges at Google with the goal to make it "easy to build simple, reliable and efficient software". [1, 2] By now the compiled and statically typed language [3] is widely used and the way it approaches network concurrency and software engineering has influenced other languages to a noticeable extent. [1] Through its structure go supports programming on various levels of abstraction. For instance, one can embed Assembler or C code into a Go program or on the other hand combine groups of components into bigger, more complex components to realize abstract design patterns. [4] Nowadays, Go is a popular choice for everything related to DevOps and therefor also for the development of command line tools. [5]

Go also tries to provide its own, official solutions for common tasks in software development. When installing Go, it comes packed alongside with a formatter (which shall ensure uniform styling across all programs written in Go) and an included suite for unit testing, just to name a few examples. Furthermore, Go supports generating documentation based on comments in the source code; much alike JavaDoc. Go features an extensive standard library which depicts a good starting point for developing your own applications. In case one wants to include a third party library, this can be done via the *go get* command. It fetches the necessary resources (usually directly from a Version Control System (VCS)) and saves the modules as a dependency in the *go.mod* file. (cite go tour and go docs)

2.1.1 Cobra

Cobra is an open source library for Go. Its aim is to provide developers with an easy way to create modern CLI applications. The cobra library is being used by noticeable projects like the CLIs for Kubernetes or for GitHub. The idea behind Cobra's intended command

schema is that commands of a well constructed CLI should read like sentences. This way, new users that are familiar with CLIs in general quickly feel native because interacting with the CLI feels more natural. In this approach, a command represents a certain action that the CLI can perform. This action than take arguments and flags to further specify on which objects and in which way the command should take action. With Cobra, one can also easily create nested subcommands. This means that a before mentioned command can also be divided into multiple sub-actions to enable detailed handling of complex actions. Further, benefits of Cobra are, among others, the automated generation of autocomplete for the most common shells as well as the ability to automatically create man pages. [6, 7]

2.2 Terraform

Modern enterprise infrastructure for software development usually makes use of cloud computing to dynamically adapt infrastructure to the fast-paced world of agile development. These clusters are usually distributed between multiple cloud services, each of which use their own individual APIs to configure their platform. Terraform is a tool that tries to minimize the effort needed to deploy these infrastructures in the long run by automating resource management as far as possible. It allows to uniformly describe the target infrastructure in an easy to learn, machine-readable definition language and automatically takes care of deploying this infrastructure at the individual Infrastructure as a Service (IaaS) providers. This approach is also known as Infrastructure as Code (IaC). With Terraform, it is also possible to save provisioned infrastructure setups as a Terraform configuration to reuse them at a later point in time or to arbitrarily extend and adapt the configuration. For configuration, either JavaScript Object Notation (JSON) or HashiCorp Configuration Language (HCL) can be used, with HCL being the preferred way by the developing company HashiCorp because of its advanced features compared to JSON. Terraform can only unleash its full potential because of cooperations with all major software and hardware vendors and providers. HashiCorp partners with over 160 companies and services the most noticeable ones being:

- AWS,
- Atlassian,

- Cloudflare,
- Google,
- Microsoft, and
- Oracle.

The most common use cases for Terraform include general IaC, managing Kubernetes (section 2.3), multi-cloud deployment, management of network infrastructures, and management of virtual machine images. Terraform additionally integrates tightly with other HashiCorp services like Vault (section 2.7).

2.3 Kubernetes

Kubernetes (often short: k8s) is an open source solution to ease up and automate management of container based services. It follows a declarative paradigm. This means that the users just needs to describe the desired state – for example through the use of configuration files or via the Kubernetes CLI – and Kubernetes determines the steps by itself which are necessary to reach and maintain this state. Kubernetes also enables users to dynamically scale their applications and services. This means that the amount of resources, that are dedicated to an application, is adapted during runtime dependent, for example, on the current number of users. Furthermore, Kubernetes can perform load balancing and redundancy between different instances of the same service. [8, 9]

One instance of a Kubernetes system is called a cluster. A Cluster is composed of multiple nodes (which usually are virtual machines or physical servers) which run the actual applications. An Application runs inside some kind of container, internally called Pod. The interaction with a cluster is managed by the so-called *Kubernetes Master*. It is a central controlling unit. The user actually never interacts with the nodes themselves directly. [10]

2.4 Gardener

Even though there are tools that help with creating and updating single Kubernetes clusters, it is rather hard to manage many clusters. This is where Gardener comes into play. It is a Kubernetes native extension that manages Kubernetes clusters as a service, packing the performance to manage up to thousands of clusters. One key concept that is applied is so called self-hosting. This means that Kubernetes components are run inside of Kubernetes clusters and is done, because Kubernetes is the go-to way to easily manage software in the cloud. [11, 12, 13, 14]

Gardener's architecture is constructed much like Kubernetes itself although not individual Pods are managed but entire clusters. The root of all is the so-called *Garden Cluster*. It is the main interface for the Gardener administrator and hosts, among others, the Garden Cluster control plane, the Gardener dashboard, the Gardener API server, the Gardener controller manager and the Gardener scheduler. Then there are two other types of clusters, *Seed Clusters* and *Shoot Clusters*. One Seed Clusters manages the control planes (thus API server, scheduler, controller manager, machine controller etc.) of its Shoot Clusters. There is at least one Seed Cluster per IaaS provider and region. The Shoot Clusters control planes are deployed as Pods inside the Seed Cluster and can therefore be created with standard Kubernetes deployment and support rolling updates. Shoot Clusters only contain worker nodes and are the Kubernetes Clusters that actually become available to the end-user and can be ordered in a declarative way. The clusters created by Gardener are vanilla Kubernetes clusters independent of the underlying cloud provider. [13, 15] (More details of the described architecture can be seen in Figure 2.1.)

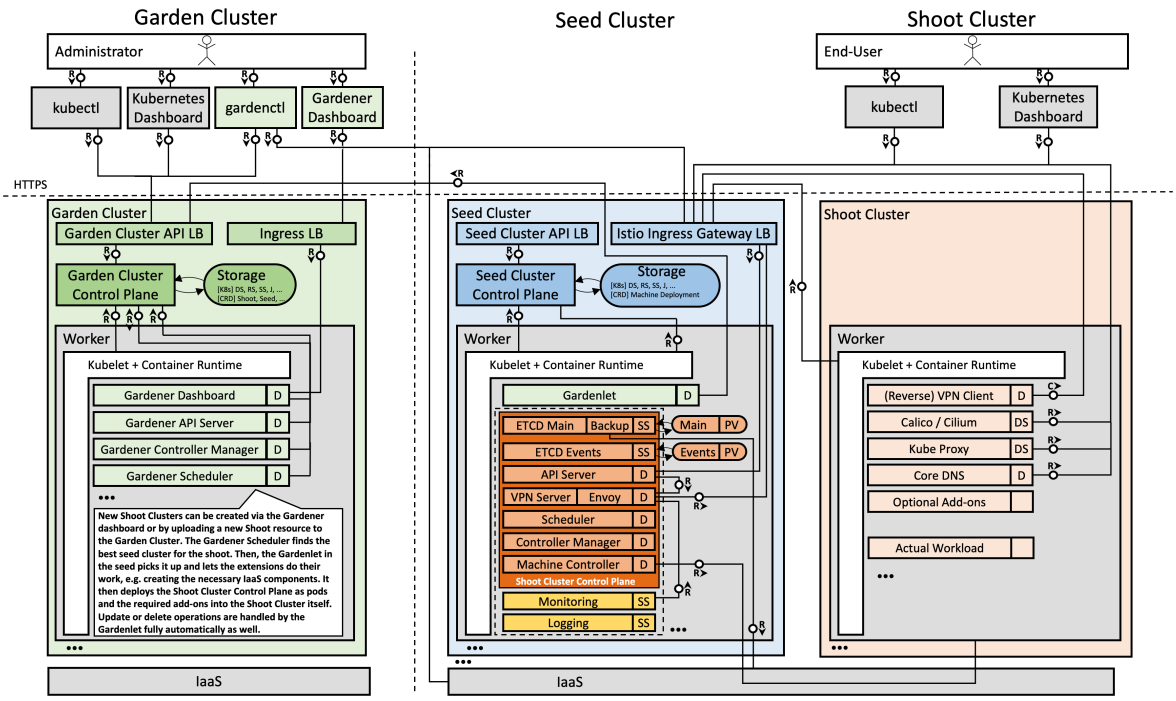

Figure 2.1: Gardener architecture [11]

2.5 AWS

AWS is a subsidiary company of Amazon that offers a cloud computing platform with various services, such as Simple Storage Service (S3) [16] or Elastic Kubernetes Service (EKS) [17]. Started in 2006, AWS nowadays runs data centers all over the world to provide scalable, reliable, and high performing services. [18, 19]

At the time of writing, AWS is the cloud provider with the biggest market share. [20] Also, most of the supervising department's clusters are currently being deployed on AWS. Because of these reasons and also due to constraints in time and complexity, the bootstrapping process worked out in this report is going to be narrowed down to deployment in an AWS environment.

2.6 Jenkins

Jenkins is a widely used open source automation server build with Java. It is a Continuous Integration and Continuous Delivery (CICD) tool aiming to save time by automating repetitive tasks like building projects, running test sets and deployment. Jenkins supports many plugins (over 1800) via its update center which give you the ability to integrate with most of the common tools in development and automate practicably any project. Jenkins can also be deployed on multiple machines to spread load and ensure quick and efficient operation. [21, 22, 23]

In context of this project, Jenkins will be used after the successful bootstrap to take care of running Terraform and performing the actual deployment.

2.7 Vault

Vault is an open source service with the primary task to provide a central control unit to manage and organize enterprise secrets. It encrypts secrets both at rest and in transit. Access to the secrets can be granted granular per user through the use of Access Control Lists (ACLs). Furthermore, Vault audits access to the secrets. That means that it keeps a detailed log on whom accessed what secret at which point in time. If there was a security breach, where an unauthorized person got access to Vault, this protocol can be used to tell, if a specific secret has been read by the attacker or if it is still safe to use.

Vault is designed to be highly pluggable. An instance is composed of *storage backends*, *audit log instances*, *authentication providers* as well as *secret backends*. Each of these can be impersonated by a variety of different components. This makes it possible to use different trusted authorities for attestation of identity. For example, among others LDAP, JWT, GitHub, and Radius can be used. An automated build service could very well use a different service to authenticate to Vault than a human user.

Secrets and encryption are often the weak spot in applications. If a secret gets leaked and the leak stays unnoticed, attackers could gain long term access to a system. As a solution, Vault offers *dynamic secrets*. When a client requests the access credentials for a supported system, Vault creates a short-lived secret just for that specific client. Because the client is only accessing Vault, it does not have to bother with key creation nor rotation and an

increased layer of security is added by not using secrets for an extended period of time. Also, if a dynamic secret gets leaked, this single secret can be revoked individually. If all clients accessing the resource used the same credentials, changing or blocking those could potentially cause an outage of the whole system.

When it comes to encryption, it can happen rather quickly that a single mistake compromises the security of the whole application. Because of this, Vault offers encryption as a service. The idea is, that Vault concentrates on the single task to handle credentials and encryption safely. The broad variety of applications have a different focus and are not developed with the necessary expertise to guarantee safe implementation of security measures. Vault, on the other hand, uses implementations that are audited by the open source community as well as independent experts. Those are then provided as a high level API to application developers. That way, the encryption process of data gets very easy while, at the same time, Vault can handle the used encryption keys directly, and they are never actually sent to the application itself. [24]

During the bootstrap, this project is about, a technical user is created for Terraform operations. An access key is created for this user and then saved to Vault. This way Jenkins then can obtain this access key and use it to run the actual cluster deployment with Terraform.

3 Conceptional Thoughts

3.1 Interaction with Services

Before implementing the bootstrap, one of the major questions is what way should be chosen for programmatically interacting with the services AWS and Vault. Both provide multiple possibilities.

3.1.1 AWS

When it comes to AWS, there are three obvious ways that could be chosen.

REST API Practically all required AWS functions can be accessed via its Hypertext Transfer Protocol (HTTP) API. Since Go's standard library natively includes a HTTP client, utilizing this would be a very lightweight solution. You would just have to instantiate a HTTP client object in Go. This object then already has all the required functionality to send requests to the API. An API call is made through a HTTP request with a specific method (GET, POST, PUT, DELETE etc.) to an API endpoint. This endpoint is specific to the operation you want to perform and represented by an Uniform Resource Locator (URL). Additional parameters and input data for the operation can be specified in a key value style via URL parameters, the request header, or the request body. URL parameters can be generated with string replacement and then appended to the base URL for the AWS API. The request body is a bit more complex to construct. It is basically a structure, that maps strings to basic data types or subordinate maps. This has to be constructed as a structure within Go and can then be encoded into a format supported by the HTTP client.

Generally, using the HTTP API would grant great flexibility because you construct all the requests on your own and therefore have detailed control over what happens without any additional layer of abstraction. On the other hand, since multiple different API calls are required, every single one of the needed calls would have to be manually constructed.

This is a lot of work, prune to errors that are hard to debug, and has a bad influence on the readability of the code in general because the API calls would get prevalent to the actual program logic.

AWS CLI The AWS CLI provides a very easy and intuitive interface to the user for interacting with AWS. Theoretically, it is intended to be explicitly installed on a system and to be used by a human user rather than programmatically. Anyway, Go natively provides the functionality to execute commands on system level. By this mean, also the AWS CLI could be used in the program.

But using the CLI would imply multiple drawbacks. CLIs often do not have a stable human interface and therefore the output returned by the CLI is subject to change. This is no good if the program has to parse the output and behave according to the results because the program could break easily and unnoticed just by updating the CLI. Although, in the special case of the AWS CLI the user can choose between several output formats including JSON notation, so a changing interface probably would not be of a problem. What is more of a concern is the fact that the CLI containing the bootstrap should be part of a container image packing various tools to work with clusters. The AWS CLI is entirely written in Python. If the AWS CLI should be used, Python would have to be installed into this container as well noticeably increasing the resulting image size. Also, when run locally, the Go application would have to rely on an existing installation of the AWS CLI to function correctly or check for its existence and prompt the user to satisfy the dependency manually in case it is missing. Just running the Go application executable would not be sufficient to perform the bootstrap. Because of these reasons, embedding the AWS CLI into the newly created Go CLI coordinating the bootstrap should be seen as a solution of last resort.

AWS Go SDK The AWS Software Development Kit (SDK) is a library provided by Amazon itself to interface its AWS services. It is not only available for Go but for a variety of different languages. To make use of it, during development it can be acquired with `go get` and imported into the program. In doing so, you specifically select the needed submodules minimizing the overhead. Then, the SDK's functions can be normally used inside the Go program.

One notable pain point is the partly ambiguous documentation, dependent on the part and version of the SDK. For instance, while the methods for user management have well documented error codes in version 1 of the SDK, telling you exactly what kind of errors you can expect, while version 2 – which supposedly does a better job on error handling – does not bother to take note on the possible error types, sometimes requiring in depth research to discover what you can expect. Luckily, errors are not of a big concern for this project, as near to all kinds of occurring errors just mean an unrecoverable program state and cannot explicitly be handled by the application itself. Furthermore, SDKs bring the inherent problem, that you completely rely on the provider in terms of update. If AWS changed their API while not touching the SDK, the program would stop working with no way to fix it rather than waiting for Amazon to update the rest of their codebase. On the other hand, since this is not a third party but an official SDK, one could also see an advantage in it. Staying with the example of the changed API and assuming that the SDK gets updated at the same time as the API, our codebase would simply continue to work. At the same time, you would be responsible to update a program utilizing the REST API entirely on your own. As, in this case, the SDK is released and maintained by Amazon their selves, and because of the introduced simplicity of working with an SDK rather than the API directly, this variant will be used in the following.

3.1.2 Vault

The aspects and options to consider for Vault are very similar to those for AWS. Vault also has an HTTP api, a CLI and an own Go SDK. One noticeable difference here is that – even though the SDK is provided by the vendor itself – the documentation for the SDK is not good at all, especially in terms of login. To get some login methods running, one has to dig rather deep. Some functionality simply is not documented at all.

But the by far more important points are, that the newly created CLI by the supervising department already contains some work in progress parts that interact with Vault. It would not make much sense to reimplement key aspects that these solutions already cover. Furthermore, mixing different approaches to access Vault would be bad practice as it makes things unnecessarily complex and introduced multiple points of failure for key functionality. Because of this, Vault's HTTP API will be used in this project. This can be additionally reasoned with the advantages of HTTP APIs described in the predeceasing

section with the difference, that there are not many functions of Vault that need to be accessed despite reading and writing secrets. The programming overhead in writing these interactions manually via the HTTP client is pretty small.

3.2 Assumptions

The before mentioned container image already contains scripts to manage login to AWS. These scripts take a Security Assertion Markup Language (SAML) response and generate the file `~/.aws/credentials` out of it. This file contains sets of AWS credentials consisting of an access key ID, a session token and a secret key. These credential sets are sufficient authentication information to interface AWS. For the following implementation, the assumption is made, that this file already exists and contains a valid key set.

3.3 Policy Management

In AWS a policy grants access to specified resources like S3 storage buckets. During the bootstrap, a technical service user is created for Terraform which needs the rights to access the Terraform state bucket. To grant these permissions, a policy is needed. There are different types of policy that can be chosen from, the two most noticeable ones being *managed policies* and *inline user policies*. A managed policy is an own entity that is created and can exist independently of a user or similar objects. To grant a user the rights that the policy grants, the policy is *attached* to the user. An inline policy on the other hand is a property of the user it is attached to. In does only apply to this single user and cannot be attached to different users, groups or roles.

The policy created in the bootstrap will only be used for the terraform user. Therefore, theoretically it would be the proper way to go to use an inline policy so that the list of other policies does not get cluttered unnecessarily. Although, different size limits apply to the different policy types. The total size of all inline policies that can be attached to a user must not exceed 2048 bytes. Because the cluster name is used multiple times in the policy, if you do the calculation, it becomes clear that the usage of inline policies would entail harsh length limits on the cluster name that would not be reasonable. Therefore, managed policies will be used even though the management has a bigger overhead.

4 Implementation

As the supervising department already works on the CLI in which the bootstrap shall be included, the following implementation will take part inside an existing Go project.

Go programs are structured into packages that logically separate different aspects and functionalities. Most of the development in this section is directly connected to interfacing with AWS and will therefore take place inside the newly created `aws` package.

The bootstrapping process will have to traverse the following steps in order:

1. create an S3 bucket as a state storage for Terraform
2. create Terraforms technical user
3. create or update the access policy for the technical user
4. attach the access policy to the technical user
5. if access keys exist, check them for expiration
6. if keys are expired or do not exist, perform a key rotation
7. safe newly generated keys to Vault

As this process is too long and complex to present it in all its details, excerpts will be used to exemplarily present the process of formation.

4.1 AWS Client Struct

A struct in Go is basically a collection of named values that forms a new type. This is used to group data together and can be somewhat compared to objects in other programming languages. Functions can be bounded to a struct type, forming methods that can perform operations on the contained data.

This principle will be used to build a custom AWS struct for performing the bootstrap. The result can be seen in Listing 4.1. On the one hand, the struct stores a general

configuration and access clients to the needed AWS services (lines 2-5). The configuration, for example, contains the information on which global region you want to operate (like `us-east-1` or `eu-central-1`). The access clients are for S3, Identity and Access Management (IAM), and Security Token Service (STS). The types of these fields are defined by their respective SDK packages. The following variables are strings needed multiple times across the different operations.

```
1  type Aws struct {
2      cfg                awsutil.Config
3      s3                 *s3.Client
4      iam                *iam.Client
5      sts                *sts.Client
6      callerAccountID    string
7      policyArn          string
8      clusterName        string
9      clusterFullName    string
10     terraformUsername  string
11     terraformPolicyName string
12     terraformBucketName string
13 }
```

Listing 4.1: AWS struct (excerpt from aws.go)

All the variables are not meant to be set by hand but through a constructor. The constructor (Listing 4.2) takes the desired region and cluster name as input parameters and ensures that the other values get set properly. First, a configuration is loaded from default values (lines 4-9). In this step, also the aforementioned credentials gets loaded into the configuration. The constructor also packs the ability to return error. For instance, errors could occur when loading the configuration (line 4). If this is the case, `err` would be something different from `nil` (line 5), the error gets logged (line 6) and returned alongside a nil value for the AWS struct (line 7) because obviously successful generation failed. After loading the configuration, the region is set and the other names are generated based on the given cluster name and region (lines 11-17). In lines 19-21, new clients are constructed for S3, IAM, STS and persisted in the struct.

The AWS account ID is for instance required to generate Amazon Resource Names (ARNs). In this project, ARNs are needed to access policies. Therefore, the account ID needs to

be obtained. The SDK's STS client has a method to get details on the identity of the calling user and returns a struct which also contains the account ID. First, the identity struct is loaded (lines 24-28) then the ID is extracted and saved to the AWS struct. As the ARN of a resource can be clearly calculated with the account ID, the resource type, and the resource name, the ARN for the policy can already be calculated and saved to the struct (line 30). The constructor then returns the finished struct.

```
1  func New(clusterRegion, clusterName string) (*Aws, error) {
2      aws := new(Aws)
3
4      cfg, err := config.LoadDefaultConfig(context.TODO())
5      if err != nil {
6          log.Sugar.Errorf("Error while loading aws config! Error: %v\n", err.Error())
7          return nil, err
8      }
9      aws.cfg = cfg
10
11     aws.cfg.Region = clusterRegion
12     aws.clusterName = clusterName
13     aws.clusterFullName = fmt.Sprintf("aws.%v.%v", clusterRegion, clusterName)
14
15     aws.terraformUsername = fmt.Sprintf("tf-%v--bot", aws.clusterFullName)
16     aws.terraformPolicyName = fmt.Sprintf("%v", aws.clusterFullName)
17     aws.terraformBucketName = fmt.Sprintf("%v", aws.clusterFullName)
18
19     aws.s3 = s3.NewFromConfig(aws.cfg)
20     aws.iam = iam.NewFromConfig(aws.cfg)
21     aws.sts = sts.NewFromConfig(aws.cfg)
22
23     // set callerAccountID and policyArn
```

```
24      callerIdentity, err := ↙
           ↳ aws.sts.GetCallerIdentity(context.TODO(), ↙
           ↳ &sts.GetCallerIdentityInput{})
25      if err != nil {
26          log.Sugar.Errorf("Error while obtaining aws caller ↙
               ↳ identity! Error: %v\n", err.Error())
27          return nil, err
28      }
29      aws.callerAccountID = *callerIdentity.Account
30      aws.policyArn = fmt.Sprintf("arn:aws:iam::%v:policy/%v", ↙
           ↳ aws.callerAccountID, aws.terraformPolicyName)
31      log.Sugar.Debug("created aws")
32
33      return aws, nil
34  }
```

Listing 4.2: Constructor for the AWS Struct (excerpt from aws.go)

4.2 Checking AWS State and Creating Objects

The aim of this bootstrap is, that all necessary objects, users, and access rights exist in AWS. Although, it is unclear, which of these might already exist. So for each of those entities, one has to check whether they exist and create them if they do not. As the process for each of these is quite similar, it would go beyond the constraints of this report to explain each of the steps in detail. Instead, this process will be explained by the means of creating the S3 storage bucket for Terraform.

As briefly outlined above, the first action must be to check whether the bucket already exists as the bucket should not be overwritten if it already existed. This is done with the SDK method `HeadBucket` (lines 2-3) which receives the desired bucket name as input. The method call returns an output and an error. For checking bucket existence, only the error is relevant and saved to the variable `err`. The output can be discarded and therefore only an underscore is written instead of a variable name.

The SDK wraps all service errors as *API errors*. To check, if and what error occurred, the error is interpreted as `smithy.APIError` (lines 6-8). After this, it can be checked

against the error types defined by the S3 SDK package (line 9). The relevant error type is `NotFound`. If this error is on hand, the bucket does not exist and has to be created (lines 11-22).

Creating an S3 bucket via the SDK is pretty simple but has a little trick to it. The default AWS location to create buckets in is *us-east-1*. If and only if a bucket shall be created in a different location, one has to specify a so called *LocationConstraint*. Because of this, the configured region is checked to select the correct SDK method call accordingly. If creating the bucket is free of errors, the method returns nil at this point, otherwise the creation error is returned (lines 24-28). If no error occurred in the first place, then this means a successful call of the *HeadBucket* method and therefore it means that the specified bucket exists. In this case, nothing happens and the method returns nil (line 33).

```
1  func (aws Aws) CreateTerraformStateBucket() error {
2      _, err := aws.s3.HeadBucket(context.TODO(), ↙
           ↳ &s3.HeadBucketInput{
3          Bucket: &aws.terraformBucketName,
4      })
5
6      if err != nil {
7          var apiErr smithy.APIError
8          if errors.As(err, &apiErr) {
9              switch apiErr.(type) {
10             case *s3types.NotFound:
11                 if aws.cfg.Region == "us-east-1" {
12                     _, err = ↙
                           ↳ aws.s3.CreateBucket(context.TODO(), ↙
                           ↳ &s3.CreateBucketInput{
13                          Bucket: &aws.terraformBucketName,
14                     })
15                 } else {
16                     _, err = ↙
                           ↳ aws.s3.CreateBucket(context.TODO(), ↙
                           ↳ &s3.CreateBucketInput{
17                          Bucket: &aws.terraformBucketName,
```

```
18                          CreateBucketConfiguration: ↙
                                ↳ &s3types.CreateBucketConfiguration{
19                               LocationConstraint: s3types. ↙
                                     ↳ BucketLocationConstraint( ↙
                                     ↳ aws.cfg.Region),
20                          },
21                      })
22                  }
23
24                  if err != nil {
25                      return err
26                  }
27
28                  return nil
29              }
30          }
31          return err
32      }
33      return nil
34  }
```

Listing 4.3: Creating the Terraform State Bucket (excerpt from aws.go)

4.3 Policy Generation from File

In chapter 3 the difference between managed and inline policies, and also why managed policies will be used, was already covered. To create a policy in AWS you simply use an SDK method similar to the one used in the predeceasing section for creating the S3 bucket. But to do so, you need to pass a so-called *policy document* to the method. This is basically a JSON string in which the access to the desired resources is specified. In case of this bootstrap, the policy document will need to contain the specific names of the needed S3 buckets. As these names are dependent on the specified cluster name, the JSON string needs to be build accordingly. To accomplish this, Go's standard library packs a functionality called *templates*. A template in Go is a string containing special symbols to mark the positions in the text that shall be replaced. These symbols are

generally indicated by double curly braces. The braces then contain a key indicating with what property they should be replaced. In the given case, just one string needs to be replaced at multiple places in the entire time. Therefore, only the symbol {{.}} is needed. The beginning of the policy file can be seen in Listing 4.4.

```
1 {
2     "Version": "2012-10-17",
3     "Statement": [
4         {
5             "Effect": "Allow",
6             "Action":  "s3:*",
7             "Resource": [
8                 "arn:aws:s3:::ai-terraform-{{.}}",
9                 "arn:aws:s3:::ai-terraform-{{.}}/*",
```

Listing 4.4: Template File for the AWS Policy (excerpt)

But the let alone existence of this template file is not sufficient to make use of it. A new method (Listing 4.5) is written, to wrap the needed method calls of Go's template package to parse the template and build the actual policy from it. First, the template string is loaded, that in this case resides in a separate file. This gives the opportunity to change the template later on without recompiling the code. This saves time and effort and also, you might not even be able to recompile because the source code is only available internally. Files can be read in Go with the use of the `iotuil` package from the default library. When calling the `ReadFile` method, the path to the file has to be given as a string and a byte slice is returned (lines 2-6). Then, a new template can be parsed from the slice (lines 7-11). During the parsing process, the template package checks for syntax errors in the template file and aborts execution if errors occur. Finally, the policy gets *executed*. To do so, the string that should substitute the placeholders and a bytes buffer has to be passed to the templates `Execute` method (lines 13-17). This buffer can then be converted to a string and is returned by the method (line 18). The returned result is now ready to be used as a policy document.

```
1 func (aws Aws) generatePolicy() (string, error) {
2     policyTemplateString, err := ioutil.ReadFile(templateFile)
3     if err != nil {
```

```
        log.Sugar.Errorf("Error while loading policy template from file %v! Error: %v\n", templateFile, err.Error())
        return "", err
    }
    policyTemplate, err := template.New("bootstrap_aws_aicore_terraform_user_policy").Parse(string(policyTemplateString))
    if err != nil {
        log.Sugar.Errorf("Error when parsing policy template! Error: %v\n", err.Error())
        return "", err
    }

    var policy bytes.Buffer
    if err := policyTemplate.Execute(&policy, aws.clusterFullName); err != nil {
        log.Sugar.Errorf("Error when executing policy template! Error: %v\n", err.Error())
        return "", err
    }
    return policy.String(), nil
}
```

Listing 4.5: Method for Policy Generation (excerpt from aws.go)

4.4 Key Rotation

Another important aspect in cloud computing, especially in the matter of long-term maintenance, is access key management. Of course, it is very important to use secure keys and keep them secret. But frequently renewal of keys may not be neglected. That way, even if an attacker gains access to a key, this will not be a problem for long. Although, this may sound simple, it has to be taken care that no legitimate user gets locked out of the systems during this process. AWS therefore supports the parallel management of two

access keys. That way, you can delete and renew a key while, in the meantime, having undisturbed access to the system with the other key. This process is called *key rotation*.

The bootstrapping procedure shall perform a key rotation if keys already exist for the technical user and a key exceeds a maximum age of seven days. The age of a key can be determined by its metadata. Listing 4.6 shows how the check is performed on whether a new key has to be created or not. The method takes a slice of access keys to check as parameter and returns a boolean that indicates if a new key has to be created in the following (line 1). Since an AWS user can have anything between none and two access keys, there are three different cases to be distinguished. The first and most basic case is when no access keys exist for a user. In this case the length of the input slice is zero and a new key has to be created (lines 2-5).

If the Terraform user has exactly one access key, there are two possibilities. Either the key is younger than seven days and nothing happens or the key exceeds the maximum age of seven days and a new key has to be created (lines 7-14). The key is not deleted in this case because if this would be the very key that is currently being used for automated access through Jenkins, this would block the service out. Instead, deleting the old key will happen the next time the bootstrapping procedure is executed. This way, there is no danger of accidentally blocking the access of a service.

If two keys exist, the older one of the two has to be determined first. Since it is known that there never ever will be more than two keys and the cases for zero keys and one key is already handled, it can safely be assumed that the first and second index of the input slice contains valid access keys. Therefore, it is very easy to determine the oldest key with a simple if statement (lines 16-19). After this, the procedure is very similar to the case with only one access key. It is checked, if the key is older than seven days (line 20). If this is the case, a new one has to be created (lines 29-30), although this time the old key gets deleted (lines 21-28). If the key is younger than these seven days, again nothing happens (lines 33-34).

```
1 func (aws Aws) checkDeleteAccessKeys(accessKeys ↙
    ↳ []iamtypes.AccessKeyMetadata) (createNewKey bool, err ↙
    ↳ error) {
2     if len(accessKeys) == 0 {
3         createNewKey = true
4         return
```

```
    }

    if len(accessKeys) == 1 {
        if time.Since(*accessKeys[0].CreateDate).Hours() > float64(maxKeyAgeInDays)*24*time.Hour.Hours() {
            createNewKey = true
            return
        }
        createNewKey = false
        return
    }

    olderKey := accessKeys[0]
    if accessKeys[1].CreateDate.Before(*olderKey.CreateDate) {
        olderKey = accessKeys[1]
    }
    if time.Since(*olderKey.CreateDate).Hours() > float64(maxKeyAgeInDays)*24*time.Hour.Hours() {
        _, err = aws.iam.DeleteAccessKey(context.TODO(), &iam.DeleteAccessKeyInput{
            AccessKeyId: olderKey.AccessKeyId,
            UserName:    olderKey.UserName,
        })
        if err != nil {
            createNewKey = false
            return
        }
        createNewKey = true
        return
    }

    createNewKey = false
    return
}
```

Listing 4.6: Key Rotation (excerpt from aws.go)

4.5 Updating Keys in Vault

Every time a new key gets created, it needs to be saved to Vault. As already mentioned in chapter 3, there is already an implementation of a basic Vault client by the supervising department that just needs to be extended to add the functionality to add keys to Vault. For this purpose, the method shown in Listing 4.7 is added to the existing Vault package.

The method binds to a config struct, storing information like the vault address and the path where the secret shall be created or where it can be accessed and satisfying a Vault client interface. The access key to be saved is passed to the function as a parameter (line 1). Then, the call to the Vault API is prepared. First, the endpoint URL is constructed from the Vault base address and the desired secret path (line 2). Then, the request body containing the new secret is prepared. Vault takes new secrets from the *data* sections of JSON bodies. Therefore, a new map is created and the passed access key is mapped to the key `data`. Go then provides a method, again from its default library, to convert an ambiguous map into a JSON bytes slice (lines 4-6). Then, a new HTTP client is constructed (line 8). Also, a new HTTP request gets build. The request will be of type `POST`, targeted to the determined URL, and utilizing the generated JSON body (line 9). The header of the request must contain a Vault token for authentication. This token is already available in the config struct and can simply be added to the request header (line 10). The HTTP client is then used to send the request (lines 11-15).

```
1  func (conf *config) CreateOrUpdateSecret(key ↙
       ↳ iamtypes.AccessKey) {
2    url := fmt.Sprintf("%s/v1/secret/data/%s", ↙
         ↳ conf.aiVaultAddress, conf.secretPath)
3
4    reqBody := make(map[string]interface{})
5    reqBody["data"] = key
6    reqBodyBytes, _ := json.Marshal(reqBody)
7
8    client := &http.Client{}
9    req, _ := http.NewRequest("POST", url, ↙
         ↳ bytes.NewBuffer(reqBodyBytes))
10   req.Header.Set("X-Vault-Token", conf.vaultToken)
11   res, err := client.Do(req)
```

```
12    if err != nil {
13      log.Sugar.Fatal(err)
14    }
15    defer res.Body.Close()
16  }
```

Listing 4.7: Saving Keys to Vault

5 Future Work

To finish off the bootstrap, work beyond the scope of this report is required. Some fundamentals and considerations regarding these tasks shall be discussed here, although their realization will not be depicted.

5.1 Establishing Tests for the Bootstrap

When it comes to testing the implemented solution, it is important to distinguish different categories of tests. On the one hand there are *unit tests* and on the other there are *integration tests*.

Unit tests are about testing small components of code for functionality in different scenarios. Do to so, the test often isolates the components from the remaining code. Dependencies on other systems are usually swapped out to create this isolation. This process is called *mocking*. In general, unit tests should not have side effects and should be completely independent of the rest of the application. Because of this, unit tests are very fast, simpler in structure and therefore also easier to write.

Integration tests on the other hand are way more complex than unit tests. Like their name suggests, they test the integration of a module with other modules. This is done when a unit test is not sufficient for testing some functions because of its isolation property. Integration test do not try to mitigate side effects but consider them from the beginning. Generally, integration tests are more complex to set up and slower than unit tests. Also, integration test might often rely on external resources which failures are beyond the control of the developer. As a result, it is usually desirable to use many unit tests and only few integration tests.

The difficulty with the implementation of the bootstrap at this state is that the interaction with AWS is deeply tied into the business logic of the bootstrap. As a result, the bootstrap

would require integration testing as a verification, although an integration test should only cover the actual interaction with AWS and not the bootstrap itself which should rather be checked with basic unit tests. To cover the business logic of the bootstrap with unit tests, the bootstrap therefore has to be decoupled from the interaction with AWS. The AWS part would then have to be covered with an integration test, although a much smaller one, which could focus solely on the third party service and would not get mixed with the departments own business logic. To do so, there are different approaches that can be chosen from.

Client Interfaces A common approach to write unit tests for something like the bootstrap would be to replace the clients of the AWS SDK that communicate with the AWS backend with mocked clients. These mocked clients could then fake the interaction with AWS and deliver reproducible results to test the actual business logic. In Go, this is generally rather easy to achieve through the use of interfaces. An interface in Go is just a definition of method headers. Any type that implements the specified methods, automatically also implements the interface. For making the actual clients interchangeable with mocked clients, interfaces would have to be specified, that define all SDK methods the clients need. Then, new types could be constructed, that implement those methods and return the desired values for mocking. Now, instead of using the types of the actual SDK clients when referring to the clients (in this case the types of the client variables in Listing 4.1), the interfaces would be used. This enables the tests to replace the clients with the mocked clients without changing the program execution, because the same methods can be called but just on different objects.

This is an easy approach if only a few methods have to be mocked. The problem in the context of this project is, that especially for the IAM client, many methods would have to be mocked individually, which is a lot of work, creates a lot of overhead for creating the interface, and it is difficult to cover all special cases and possible errors.

Function Pointers In some ways, this approach is rather similar to the aforementioned one. In Go, functions can be stored in variables just like anything else. So by extracting certain functionality into individual functions, instead of creating entire mock clients, just some functions could be swapped out for other functions delivering the mocked results. To make use of this principle, another layer of abstraction would need to be implemented

that wraps the SDK methods into package scoped functions, and, if feasible, aggregates multiple SDK methods into one wrapper. These wrappers could be referred to with function pointers. For unit testing, only these pointers would have to be adjusted to point to the mocked methods. This can happen directly in the test file as in Go the tests are located in the same package. The wrappers themselves could be verified for functionality with integration tests.

Although this approach reduces the complexity of the actual mocking by omitting the use of interfaces and rebuilding the entire clients, as a downside it would require some restructuring to the code because the calls to the SDK methods would have to be replaced with the function pointers.

5.2 Integration with the CLI

Eventually, when the external structure of the bootstrap package becomes final, the functionality provided by the newly created package still needs to be included into the cobra CLI. Thanks to the default structure of the cobra framework, this process is very simple. In cobra, each interaction with the application is contained in a command. All these commands, despite the root command which is the app itself (in this case called `ictl`), have exactly one parent command, origin either directly or indirectly from the root command and optionally run an action. Command definitions are normally located in a dedicated package. Adding a command is as easy as adding a struct variable of the type `Command` defined by the cobra package. The most important fields of a command are `Use`, which defines the name with which the command is called, and `Run`, which is a Go function that is run when the command is called. Additional input values for the action can be provided by the use of *flags*. Those are defined in an init function of the commands package using the command's flag methods provided by the cobra framework.

For this bootstrap, probably a dedicated subcommand like `bootstrap` should be added that is a child of the root command. The bootstrap would then be called by typing `ictl bootstrap` in the terminal. The run method would then internally call the functions exported by the package containing the bootstrap logic. Information required by the bootstrap (region and cluster name) could be realized as required flags of the bootstrap command and then passed to the bootstraps functions inside the run method.

To offer the possibility to add bootstrapping processes for different cloud providers at a later point in time, the AWS bootstrap could also be implemented as a subcommand for the bootstrap command in the first place. The only difference in realization would be that the bootstrap functions are not called in the run method of the `bootstrap` command but in the run method of the child command `aws`. Eventually, performing an AWS bootstrap could look like
`ictl bootstrap aws --name clusterName --region eu-central-1`.

6 Evaluation

Literaturverzeichnis

[1] Pike, R. *Using Go at Google - The Go Programming Language*. 08/2020. URL: https://go.dev/solutions/google/ (visited on 08/02/2022).

[2] *The Go programming language — GitHub*. URL: https://github.com/golang/go (visited on 08/02/2022).

[3] Chris, K. *What is Go? Golang Programming Language Meaning Explained*. 10/2021. URL: https://www.freecodecamp.org/news/what-is-go-programming-language/ (visited on 08/02/2022).

[4] Maurer, C. *Objektbasierte Programmierung mit Go*. Springer Vieweg, 2021.

[5] Van Sickle, M. *Go: Getting Started — Pluralsight*. 01/2020. URL: https://app.pluralsight.com/library/courses/getting-started-with-go/table-of-contents (visited on 08/02/2022).

[6] *A Commander for modern Go CLI interactions — GitHub*. URL: https://github.com/spf13/cobra (visited on 08/02/2022).

[7] *Cobra. Dev*. URL: https://cobra.dev/ (visited on 08/02/2022).

[8] Bloß, A. *Containerorchestrierung mit Kubernetes - Teil 4 — x-cellent technologies GmbH Blog*. 2019. URL: https://www.x-cellent.com/blog/containerorchestrierung-mit-kubernetes-teil-4/ (visited on 07/15/2021).

[9] *Was ist Kubernetes? — Kubernetes*. URL: https://kubernetes.io/de/docs/concepts/overview/what-is-kubernetes/%7B%5C#%7Dwas-bedeutet-kubernetes-k8s (visited on 07/15/2021).

[10] *Konzepte — Kubernetes*. URL: https://kubernetes.io/de/docs/concepts/%7B%5C_%7Dprint/ (visited on 07/15/2021).

[11] *gardener/architecture.md at master · gardener/gardener*. URL: https://github.com/gardener/gardener/blob/master/docs/concepts/architecture.md (visited on 08/15/2022).

[12] *Gardener - The Kubernetes Botanist — Kubernetes*. URL: https://kubernetes.io/blog/2018/05/17/gardener/ (visited on 08/15/2022).

[13] *Gardener Project Update — Kubernetes.* URL: https://kubernetes.io/blog/2019/12/02/gardener-project-update/ (visited on 08/15/2022).

[14] *Gardener.* URL: https://gardener.cloud/ (visited on 08/15/2022).

[15] *Was ist SAP Gardener?* URL: https://www.datacenter-insider.de/was-ist-sap-gardener-a-955022/ (visited on 08/15/2022).

[16] *Amazon Simple Storage Service S3 – Cloud Online-Speicher.* URL: https://aws.amazon.com/de/s3/ (visited on 08/15/2022).

[17] *Verwalteter Kubernetes-Service – Amazon EKS – Amazon Web Services.* URL: https://aws.amazon.com/de/eks/ (visited on 08/15/2022).

[18] *Amazon Web Services (AWS): Über uns — LinkedIn.* URL: https://www.linkedin.com/company/amazon-web-services/about/ (visited on 08/15/2022).

[19] *Was ist Amazon Web Services (AWS)? - Definition von WhatIs.com.* 2014. URL: https://www.computerweekly.com/de/definition/Amazon-Web-Services-AWS (visited on 08/15/2022).

[20] Kumar, R. *AWS Market Share 2022: How Far It Rules the Cloud Industry?* 05/2022. URL: https://www.wpoven.com/blog/aws-market-share/ (visited on 08/15/2022).

[21] *Jenkins.* URL: https://www.jenkins.io/ (visited on 08/15/2022).

[22] *Jenkins automation server — GitHub.* URL: https://github.com/jenkinsci/jenkins (visited on 08/15/2022).

[23] *What is CI/CD? — GitLab.* URL: https://about.gitlab.com/topics/ci-cd/ (visited on 08/15/2022).

[24] *Vault by HashiCorp.* URL: https://www.vaultproject.io/ (visited on 08/15/2022).