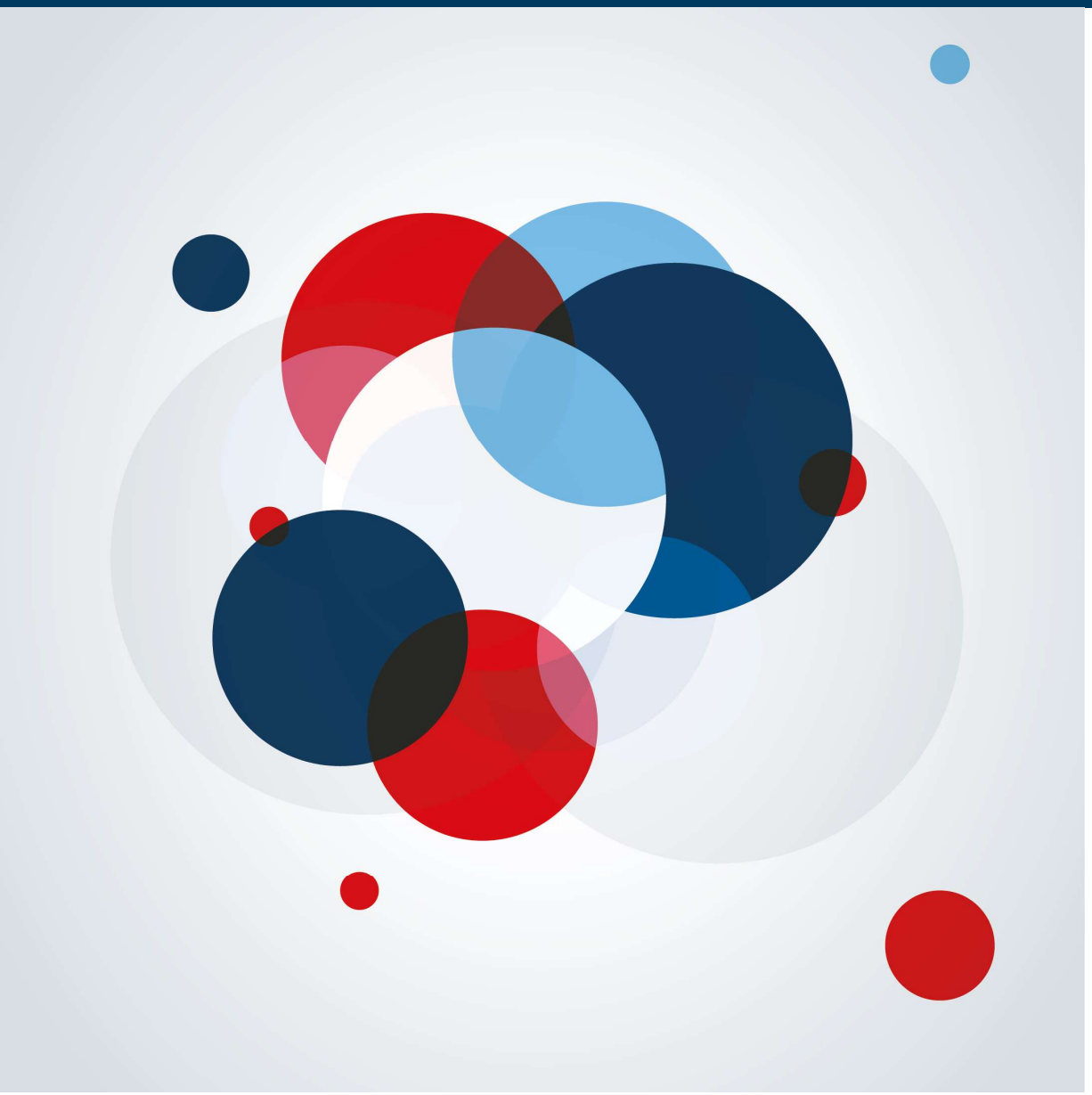


Collaborative Working and IaC

Working with Azure and Terraform

Fabian Janßen & Merlin Rothhardt, 28.07.2021

direktgruppe

“Rules of the game“

- **Ask your questions as they arise, no question is a stupid question!**
  (but please understand that at some point we will need to cut discussions)
- **Have your Smartphone at hand – we will have quizzes / feedback rounds**
- **Presentation slides will be provided to you after the training**

direktgruppe

Overview

1. **Who we are**
2. **Expectations**
3. **Resource creation in Azure – Portal & CLI**
4. **Resource creation in Azure – Terraform**
5. **Collaborative working with Terraform**
6. **Feedback & QnA**

Who we are

Merlin Rothhardt
Circle Development

Hamburg

For ~ 6 years at Direkt Gruppe

Circle Lead Java Development /
Senior Software Developer

Java Development, DevOps,
Kubernetes, Container, Azure

direktgruppe

Fabian Janßen

Circle Development

Cologne

For ~ 5 years at Direkt Gruppe

Expert Developer / DevOps Engineer

DevOps, Terraform, Kubernetes, Container, Azure

Overview -
What are your expectations?

Scan the QR-Code displayed on the left or open the following link:

https://forms.office.com/r/pKZ0BVe8Zq

direktgruppe

Azure basics

Azure basics - Ordering a Virtual Machine

In the OnPremise Datacenter

- Open a ticket / Call somebody
- Wait for the ticket to be dispatched
- Wait for the actual provisioning to finish (can be automated)
- Developers can manage the OS level, but they’re not eligible to change infrastructure (such as VM size)
- How long does it take to provision 1 VM vs provisioning 10 VMs

Identitfy your demand

Submit request

Wait for provisioning: 1 day - several weeks

In the Cloud

- Write a template with your advanced specification
- Have a review approval step (optional)
- Wait for the automated provisioning of your demand
- You can manage everything including infrastructure as well as the OS level. Adjustments can be specified in your template
- Highly flexible no matter how many resources you request*

Identify your demand

Have a template

Speed up provisioning ~30min

* Limitations may apply depending on your subscription type

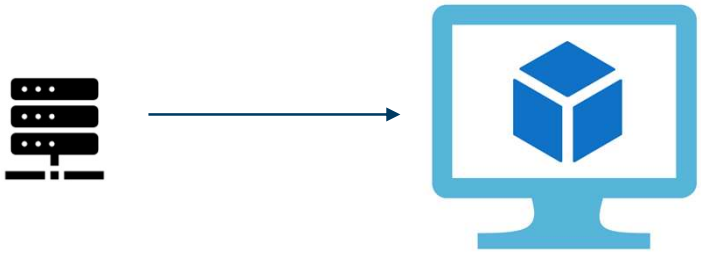
direktgruppe

Azure basics – Benefits

What to expect from moving to the cloud

- Manage your infrastructure as part of your project (Infrastructure as Code)
- Faster provisioning
- Advanced collaborative working mode
- High availability of resources
- Advanced Services and solutions
- Cost efficiency
- Easy PoC development at speed
- Improved maintainability
- Global scalability

Azure basics - Lift and Shift



Moving to the cloud

- Lift and Shift describes the process of moving an existing virtual machine to a new infrastructure like the cloud
- To get most out of the cloud, a simple Lift and Shift scenario is mostly not the best way
- Replacing parts of your software with cloud components might be the better solution for the future
- Central Cloud Concepts
  - High Availability
  - Elasticity
  - Scalability
  - Agility
  - Disaster Recovery

Different types of cloud services

Infrastructure
as a Service

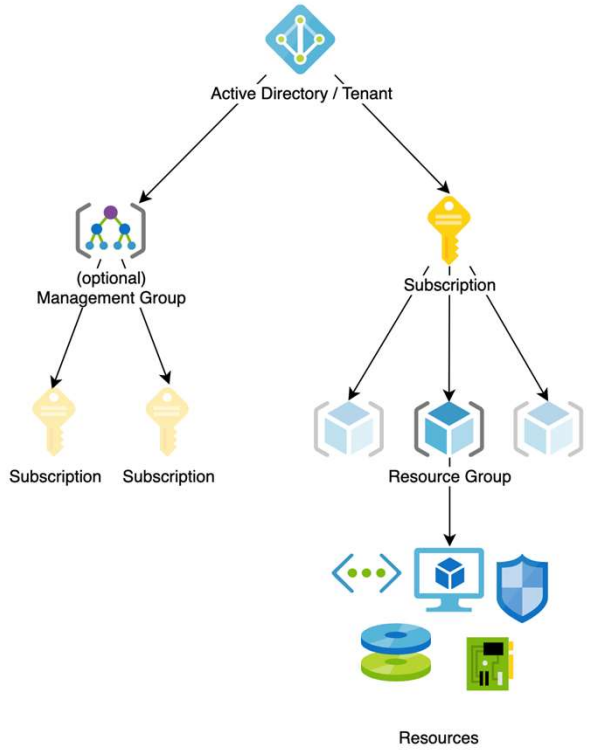
Platform
as a Service

Software
as a Service

IaaS, PaaS and SaaS

- IaaS components are **essential compute, storage, and networking resources** such as virtual machines, virtual networks, data disks etc.
- PaaS components mostly **abstract infrastructure and provide middleware** to operate your cloud demand. Such as Azure SQL Databases, Cosmos DB, Azure Functions / Web Apps
- SaaS components are **cloud based applications** that can are rented by an organization. They do not specify any resource requirements and mostly provide a web portal. In Azure, these are components like Azure Active Directoy, Key Vault or Azure Data Factory

Component Structure in Azure

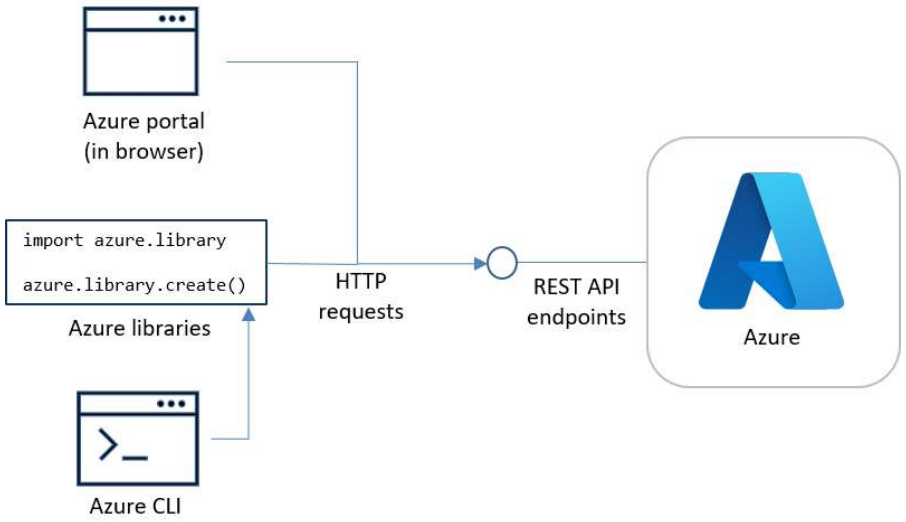


About Tenants, Subscriptions and Resources

- To operate Azure Cloud, a **Microsoft Tenant** is required. A tenant is the identity access management resource
- a subscription refers to the logical entity that provides entitlement to **deploy and consume Azure resources**
- A resource group is a **container that holds (related) resources** for an Azure solution.
- Resources are the actual billable services that can be consumed in the cloud (IaaS, PaaS, SaaS solutions)

Resource creation in Azure (Portal, CLI)

Resource creation in Azure



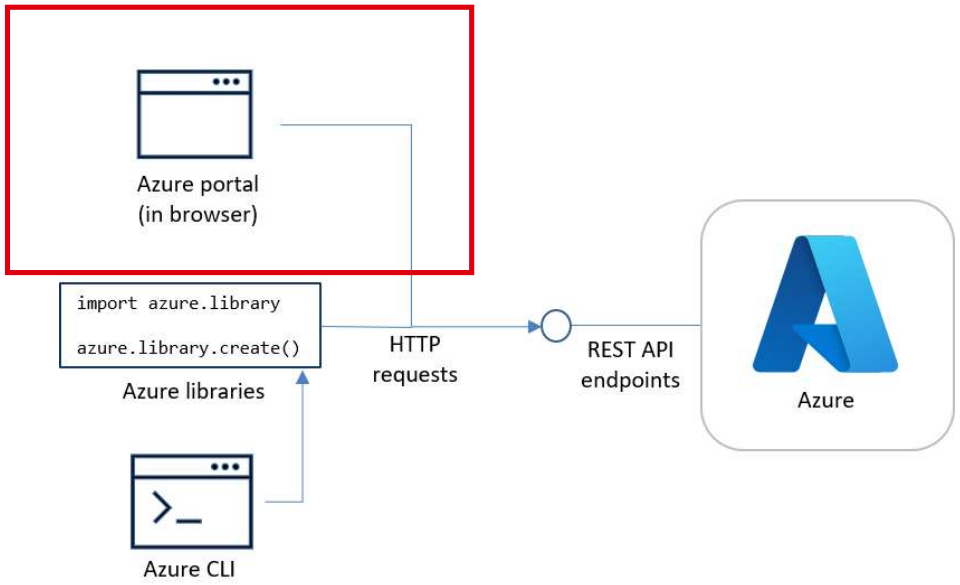
Different tools, same effect

- The central component to manage resources in the cloud is called **Azure Resource Manager (ARM)**
- ARM is providing REST endpoints to execute operations
- To facilitate integration of the ARM interface, different tools are available
  - Azure Portal
  - AZ CLI / Powershell Modules
  - SDKs
  - `azurerm` Terraform-Provider

direktgruppe

Resource creation in Azure - Portal

Live Demo

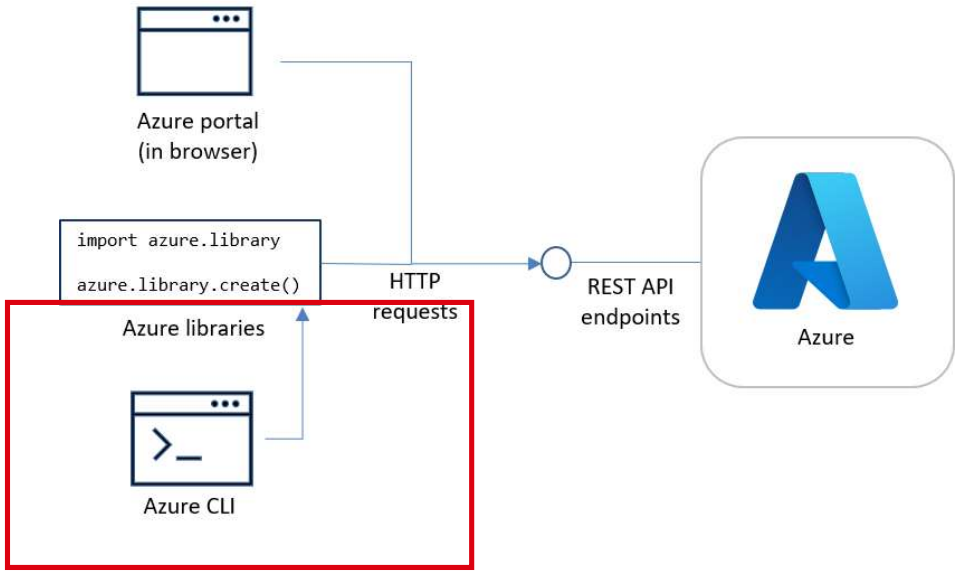


Different tools, same effect

- The central component to manage resources in the cloud is called **Azure Resource Manager (ARM)**
- ARM is providing REST endpoints to execute operations
- To facilitate integration of the ARM interface, different tools are available
  - Azure Portal
  - AZ CLI / Powershell Modules
  - SDKs
  - `azurerm` Terraform-Provider

direktgruppe

Resource creation in Azure - CLI

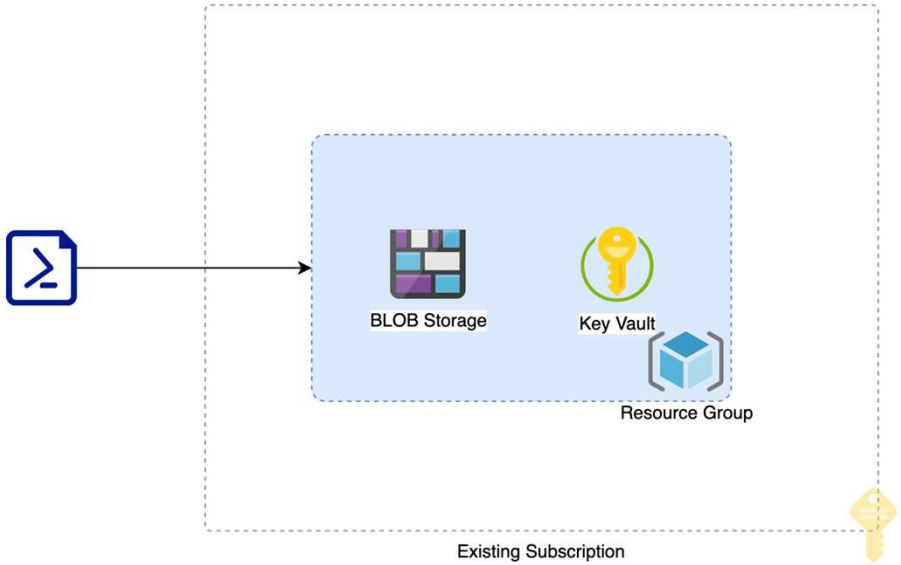


Different tools, same effect

- The central component to manage resources in the cloud is called **Azure Resource Manager (ARM)**
- ARM is providing REST endpoints to execute operations
- To facilitate integration of the ARM interface, different tools are available
  - Azure Portal
  - AZ CLI / Powershell Modules
  - SDKs
  - `azurerm` Terraform-Provider

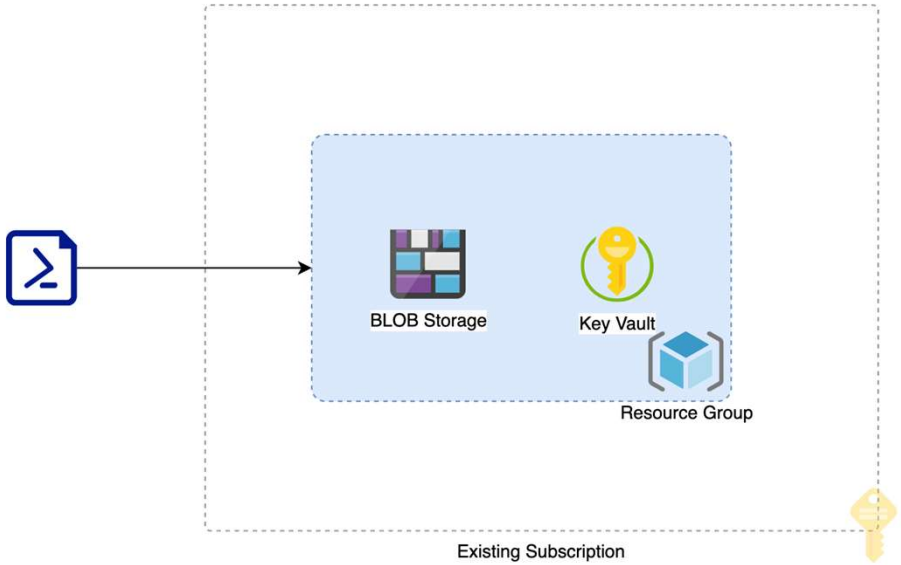
Resource creation in Azure - CLI

Different tools, same effect: Azure CLI



direktgruppe

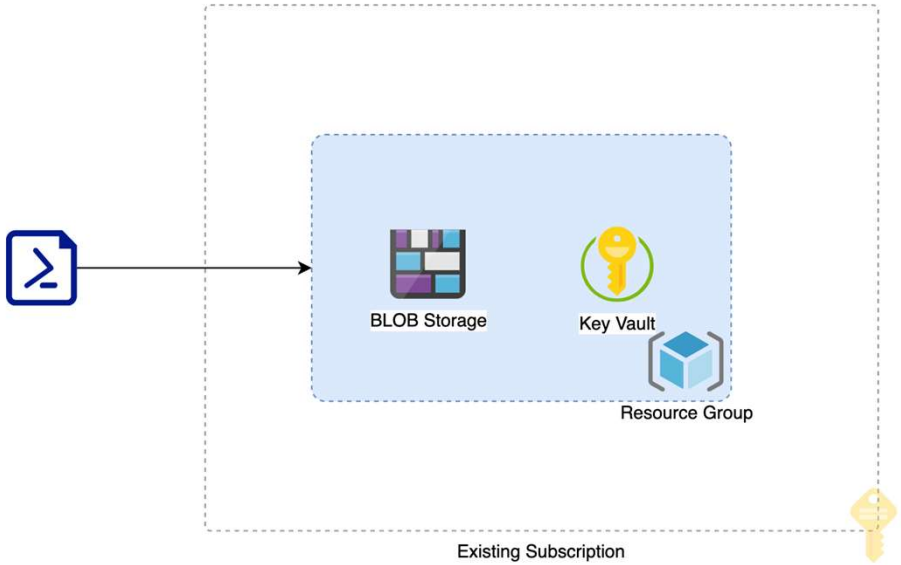
Resource creation in Azure - CLI



Different tools, same effect: Azure CLI

```
1   #! /bin/sh
2
3   # login
4   az login
```

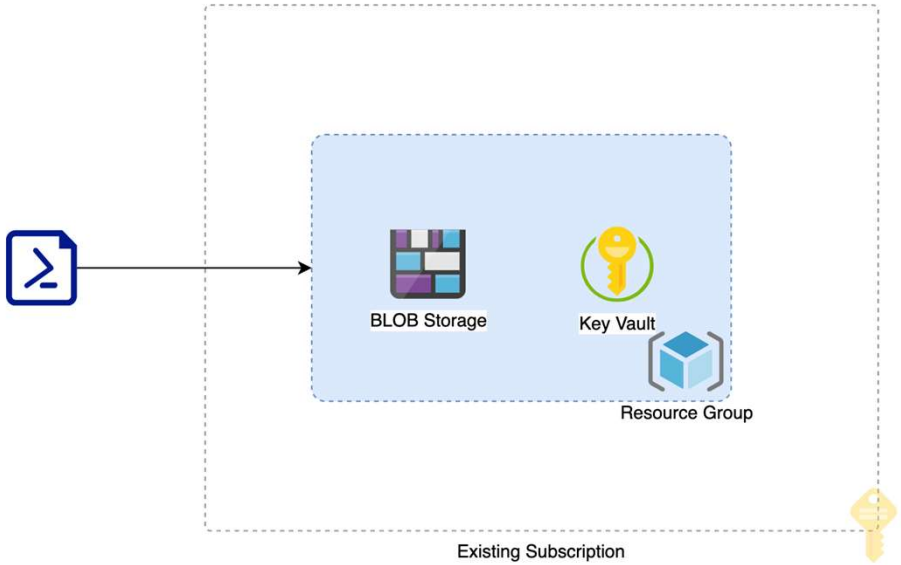
Resource creation in Azure - CLI



Different tools, same effect: Azure CLI

```
1   #! /bin/sh
2
3   # login
4   az login
5
6   # set variables
7   SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
8   RESOURCE_GROUP=contoso
9
10  # select subscription
11  az account set -s $SUBSCRIPTION_ID
```

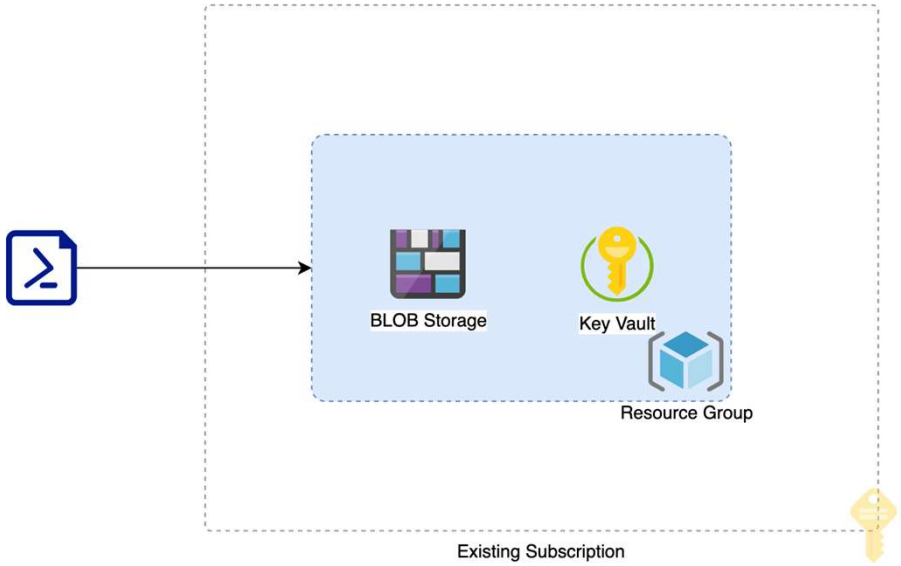
Resource creation in Azure - CLI



Different tools, same effect: Azure CLI

```
1   #! /bin/sh
2
3   # login
4   az login
5
6   # set variables
7   SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
8   RESOURCE_GROUP=contoso
9
10  # select subscription
11  az account set -s $SUBSCRIPTION_ID
12
13  # Create a resource group
14  az group create -n $RESOURCE_GROUP -l westeurope
```

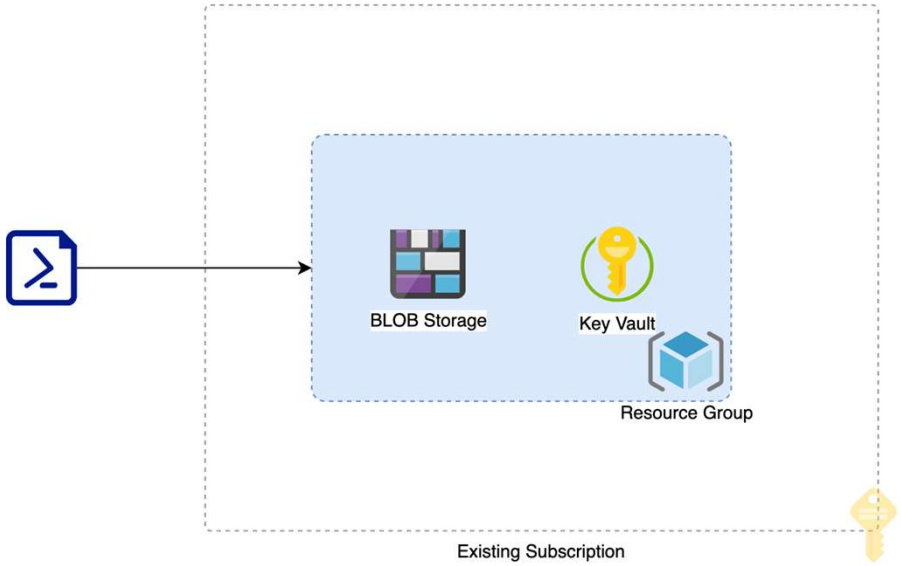
Resource creation in Azure - CLI



Different tools, same effect: Azure CLI

```
1   #! /bin/sh
2
3   # login
4   az login
5
6   # set variables
7   SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
8   RESOURCE_GROUP=contoso
9
10  # select subscription
11  az account set -s $SUBSCRIPTION_ID
12
13  # Create a resource group
14  az group create -n $RESOURCE_GROUP -l westeurope
15
16  # Create storage account
17  az storage account create -n fgrsabss123 -g
    $RESOURCE_GROUP -l westeurope
```

Resource creation in Azure - CLI

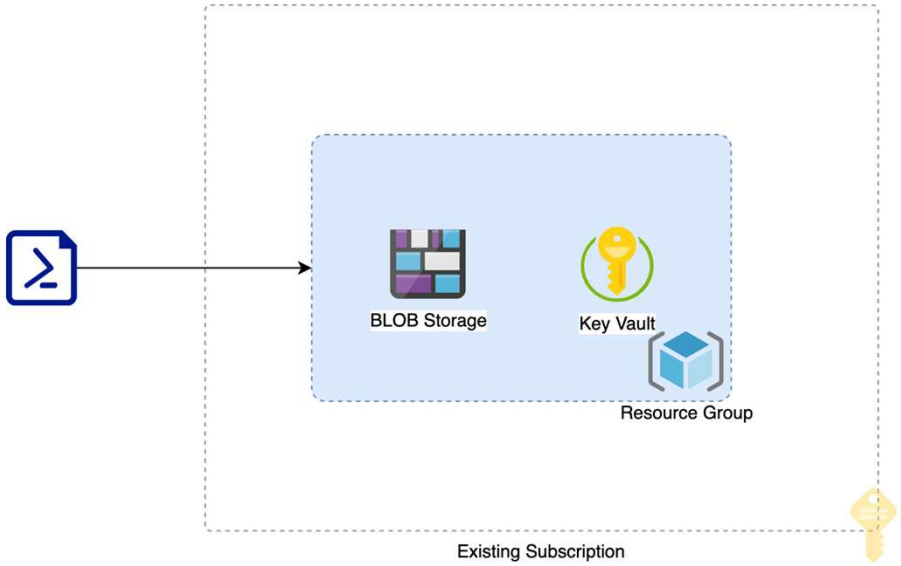


Different tools, same effect: Azure CLI

```
#! /bin/sh

# login
az login

# set variables
SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
RESOURCE_GROUP=contoso

# select subscription
az account set -s $SUBSCRIPTION_ID

# Create a resource group
az group create -n $RESOURCE_GROUP -l westeurope

# Create storage account
az storage account create -n fgrsabss123 -g
$RESOURCE_GROUP -l westeurope

# Create key vault
az keyvault create -l westeurope -n fgr-bss-ws-1 -g
$RESOURCE_GROUP
```

Resource creation in Azure - CLI

Live Demo



Different tools, same effect: Azure CLI

```
1   #! /bin/sh
2
3   # login
4   az login
5
6   # set variables
7   SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
8   RESOURCE_GROUP=contoso
9
10  # select subscription
11  az account set -s $SUBSCRIPTION_ID
12
13  # Create a resource group
14  az group create -n $RESOURCE_GROUP -l westeurope
15
16  # Create storage account
17  az storage account create -n fgrsabss123 -g
    $RESOURCE_GROUP -l westeurope
18
19  # Create key vault
20  az keyvault create -l westeurope -n fgr-bss-ws-1 -g
    $RESOURCE_GROUP
```


direktgruppe

Resource creation in Azure - CLI

```
1   #! /bin/sh
2
3   # login
4   az login
5
6   # set variables
7   SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
8   RESOURCE_GROUP=alberta
9
10  # select subscription
11  az account set -s $SUBSCRIPTION_ID
12
13  # Create a resource group
14  az group create -n $RESOURCE_GROUP -l westeurope
15
16  # Create storage account
17  az storage account create -n fgrsabss123 -g
    $RESOURCE_GROUP -l westeurope
18
19  # Create key vault
20  az keyvault create -l westeurope -n fgr-bss-ws-1 -g
    $RESOURCE_GROUP
```

What happens, if the resource group is renamed?

Scan the QR-Code displayed above or open the following link:

https://forms.office.com/r/bPeUDjY3ec

direktgruppe

Resource creation in Azure - CLI

```
The storage account named fgrsabss123 already exists under th
e subscription.
(VaultAlreadyExists) The vault name 'fgr-bss-ws-1' is already
 in use. Vault names are globaly unique so it is possible tha
t the name is already taken. If you are sure that the vault n
ame was not taken then it is possible that a vault with the s
ame name was recently deleted but not purged after being plac
ed in a recoverable state. If the vault is in a recoverable s
tate then the vault will need to be purged before reusing the
 name. For more information on soft delete and purging a vaul
t follow this link https://go.microsoft.com/fwlink/?linkid=21
47740.
```

What happened?

- az command line tool does not maintain a state or history of resources. **It just applies commands**
- A new resource group **can be created**
- Storage Account and key vault **can not be created** as their names must be unique

→ Two resources with the same unique name would have been created

Resource creation in Azure - CLI

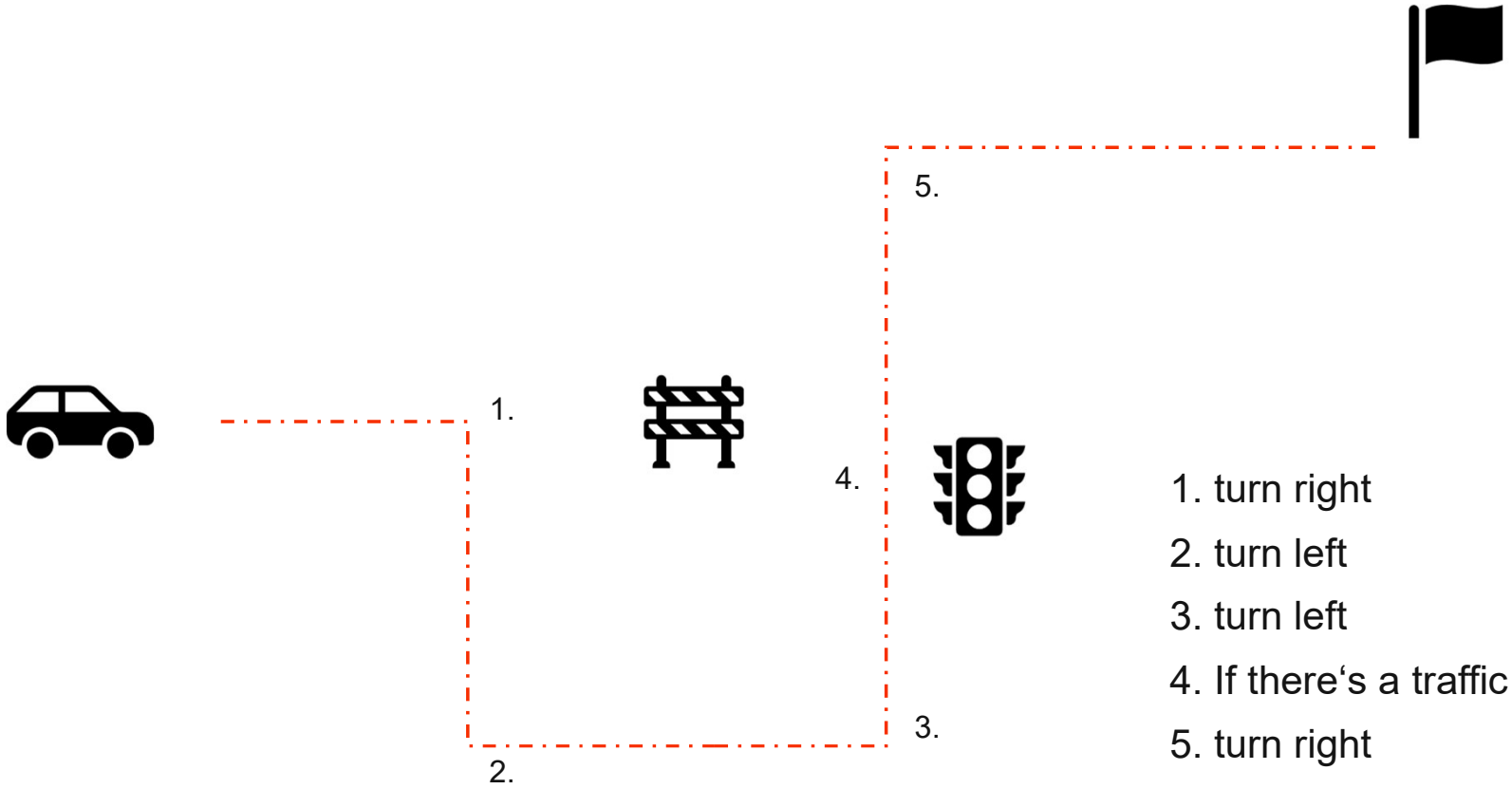
```
1   #! /bin/sh
2
3   # login
4   az login
5
6   # set variables
7   SUBSCRIPTION_ID=f6d8484f-25a6-4f36-af96-ad8071646d2b
8   RESOURCE_GROUP=contoso
9   NEW_RESOURCE_GROUP=alberta
10
11  # select subscription
12  az account set -s $SUBSCRIPTION_ID
13
14  # Create a resource group
15  az group create -n $NEW_RESOURCE_GROUP -l westeurope
16
17  # Move storage account
18  az storage account delete -n fgrsabss123 -g
    $RESOURCE_GROUP
19  az storage account create -n fgrsabss123 -g
    $NEW_RESOURCE_GROUP -l westeurope
20
21  # Move Key Vault
22  az keyvault delete -n fgr-bss-ws-1 -g $RESOURCE_GROUP
23  az keyvault create -l westeurope -n fgr-bss-ws-1 -g
    $NEWRESOURCE_GROUP
```

Change management can be automated, but …

- When a resource should be moved, we could possibly create a script that is **actively deleting and recreating** resources
- But: Every possible situation must be considered to ensure that the script is not breaking
- There is no lifecycle of a component or provisioning state maintained by CLI tools
- This approach is called **imperative programming**

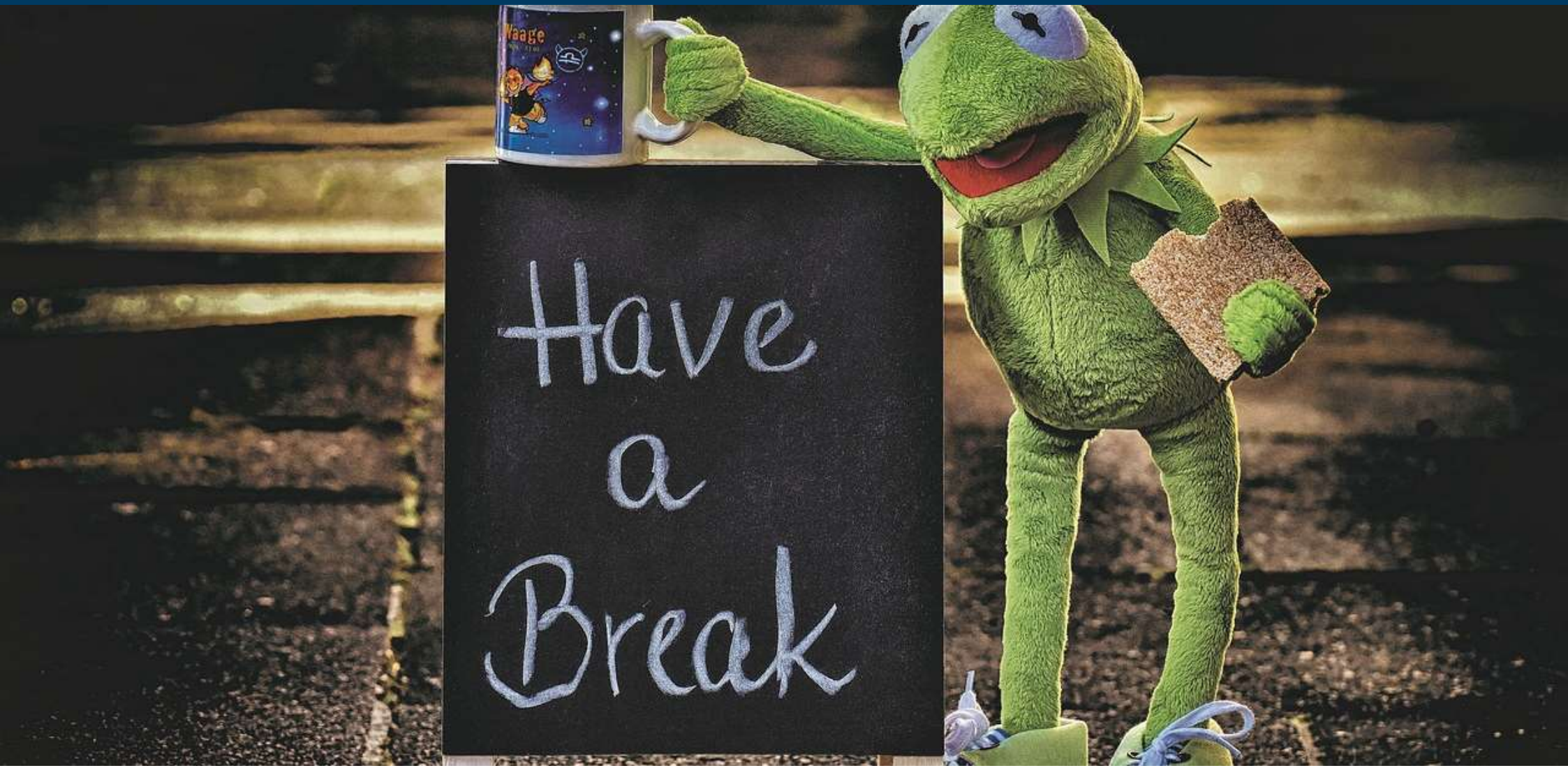
- **… and is highly discouraged ;)**

Resource creation in Azure – Imperative programming

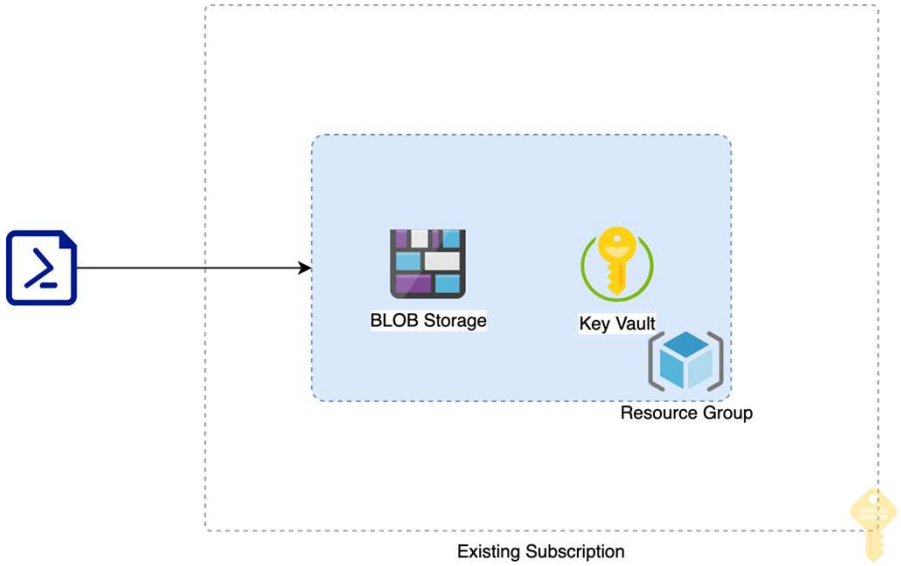


1. turn right
2. turn left
3. turn left
4. If there‘s a traffic light: wait
5. turn right

direktgruppe


Waage
Have
a
Break

Resource creation in Azure – Terraform



Idea: Create a desired state

- Instead of defining the exact commands, create a dependency structure
- Different Cloud Providers have different ways to resolve this:
  - AWS: CloudFormation
  - Azure: Azure Resource Manager
  - GCP: Cloud Deployment Manager
- One-Size-Fits-All Principle : **Terraform**

Resource creation in Azure – Terraform

```
1   provider "azurerm" {
2     skip_provider_registration = true
3     features {}
4   }
```

Idea: Create a desired state

- Terraform works with **providers**
- A provider is a **plugin for terraform to interact with remote systems**
  - Cloud Automation: Azure, AWS, GCP, Alibaba
  - Container Orchestration: Docker, Kubernetes, Helm etc.
  - Database providers
  - You can even build custom providers to interact with your system

Resource creation in Azure – Terraform

```
1  provider "azurerm" {
2    skip_provider_registration = true
3    features {}
4  }
5
6  resource "azurerm_resource_group" "resource_group" {
7    name     = "contoso"
8    location = "westeurope"
9  }
```

Idea: Create a desired state

- Entities that are created with terraform are called **resources**
- Resources are always part of a certain provider

Resource creation in Azure – Terraform

```
provider "azurerm" {
  skip_provider_registration = true
  features {}
}

resource "azurerm_resource_group" "resource_group" {
  name     = "contoso"
  location = "westeurope"
}

resource "azurerm_storage_account" "storage_account" {
  name                     = "fgrbssexample2"
  resource_group_name      = azurerm_resource_group.resource_group.name
  location                 = azurerm_resource_group.resource_group.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Idea: Create a desired state

- Entities that are created with terraform are called **resources**
- Resources are always part of a certain provider
- Resources can create dependencies to each other

- Terraform builds dependency trees and can decide what resources lead to **deletion / recreation** of depending resources

Resource creation in Azure – Terraform

```
1   provider "azurerm" {
2     skip_provider_registration = true
3     features {}
4   }
5
6   resource "azurerm_resource_group" "resource_group" {
7     name     = "contoso"
8     location = "westeurope"
9   }
10
11  resource "azurerm_storage_account" "storage_account" {
12    name                     = "fgrbssexample2"
13    resource_group_name      = azurerm_resource_group.resource_group.name
14    location                 = azurerm_resource_group.resource_group.location
15    account_tier             = "Standard"
16    account_replication_type = "LRS"
17  }
```

Idea: Create a desired state

- Entities that are created with terraform are called **resources**
- Resources are always part of a certain provider
- Resources can create dependencies to each other

- Terraform builds dependency trees and can decide what resources lead to **deletion / recreation** of depending resources

Resource creation in Azure – Terraform

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.resource_group will be created
  + resource "azurerm_resource_group" "resource_group" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "contoso"
    }

  # azurerm_storage_account.storage_account will be created
  + resource "azurerm_storage_account" "storage_account" {
      + access_tier                     = (known after apply)
      + account_kind                    = "StorageV2"
      + account_replication_type        = "LRS"
      + account_tier                    = "Standard"
      + allow_blob_public_access        = false
      + enable_https_traffic_only       = true
      + id                              = (known after apply)
      + is_hns_enabled                  = false
      + large_file_share_enabled        = (known after apply)
      + location                        = "westeurope"
      + min_tls_version                 = "TLS1_0"
      + name                            = "fgrbssexample2"
      + nfsv3_enabled                   = false
      + primary_access_key              = (sensitive value)
      + primary_blob_connection_string  = (sensitive value)
      + primary_blob_endpoint           = (known after apply)
      + primary_blob_host               = (known after apply)
```

Terraform creates a plan first

- Before applying changes, terraform displays the exact operations that are going to be performed
- These changes can be checked and approved
- Once approved, terraform runs the plan and creates / updates / deletes resources
- With each apply, terraform maintains a **.tfstate file** that identifies the resources that are controlled by terraform
- Without this .tfstate file, terraform is **not able to identify** changes on existing resources
- This is an intended **safety mechanism** in order not to accidentally break an existing infrastructure

Resource creation in Azure – Terraform

Live Demo

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.resource_group will be created
  + resource "azurerm_resource_group" "resource_group" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "contoso"
    }

  # azurerm_storage_account.storage_account will be created
  + resource "azurerm_storage_account" "storage_account" {
      + access_tier                     = (known after apply)
      + account_kind                    = "StorageV2"
      + account_replication_type        = "LRS"
      + account_tier                    = "Standard"
      + allow_blob_public_access        = false
      + enable_https_traffic_only       = true
      + id                              = (known after apply)
      + is_hns_enabled                  = false
      + large_file_share_enabled        = (known after apply)
      + location                        = "westeurope"
      + min_tls_version                 = "TLS1_0"
      + name                            = "fgrbssexample2"
      + nfsv3_enabled                   = false
      + primary_access_key              = (sensitive value)
      + primary_blob_connection_string  = (sensitive value)
      + primary_blob_endpoint           = (known after apply)
      + primary_blob_host               = (known after apply)
```

Terraform creates a plan first

- Before applying changes, terraform displays the exact operations that are going to be performed
- These changes can be checked and approved
- Once approved, terraform runs the plan and creates / updates / deletes resources
- With each apply, terraform maintains a **.tfstate file** that identifies the resources that are controlled by terraform
- Without this .tfstate file, terraform is **not able to identify** changes on existing resources
- This is an intended **safety mechanism** in order not to accidentally break an existing infrastructure

Resource creation in Azure – Terraform

```
An execution plan has been generated and is shown below.
Resource actions are indicated with the following symbols:
  + create

Terraform will perform the following actions:

  # azurerm_resource_group.resource_group will be created
  + resource "azurerm_resource_group" "resource_group" {
      + id       = (known after apply)
      + location = "westeurope"
      + name     = "contoso"
    }

  # azurerm_storage_account.storage_account will be created
  + resource "azurerm_storage_account" "storage_account" {
      + access_tier                     = (known after apply)
      + account_kind                    = "StorageV2"
      + account_replication_type        = "LRS"
      + account_tier                    = "Standard"
      + allow_blob_public_access        = false
      + enable_https_traffic_only       = true
      + id                              = (known after apply)
      + is_hns_enabled                  = false
      + large_file_share_enabled        = (known after apply)
      + location                        = "westeurope"
      + min_tls_version                 = "TLS1_0"
      + name                            = "fgrbssexample2"
      + nfsv3_enabled                   = false
      + primary_access_key              = (sensitive value)
      + primary_blob_connection_string  = (sensitive value)
      + primary_blob_endpoint           = (known after apply)
      + primary_blob_host               = (known after apply)
```

What happens, if the same terraform script applies changes without having a .tfstate file

Scan the QR-Code displayed or open the following link:

https://forms.office.com/r/KjRwAyBr5Z

direktgruppe

Resource creation in Azure – Terraform

```
Plan: 2 to add, 0 to change, 0 to destroy.

Do you want to perform these actions?
  Terraform will perform the actions described above.
  Only 'yes' will be accepted to approve.

  Enter a value: yes

azurerm_resource_group.resource_group: Creating...

Error: A resource with the ID "/subscriptions/f6d8484f-25a6-4f36-af96-ad8071
646d2b/resourceGroups/contoso" already exists - to be managed via Terraform
this resource needs to be imported into the State. Please see the resource d
ocumentation for "azurerm_resource_group" for more information.

  on main.tf line 6, in resource "azurerm_resource_group" "resource_group":
   6: resource "azurerm_resource_group" "resource_group" {
```

What happened?

- Terraform didn’t foresee that the resource already existed
- As there was no **.tfstate** file to identify the resource, Terraform tries to create it
- Two resource groups with the same name in the same subscription would have been created which led to an error

Resource creation in Azure – Terraform advanced example

Live Demo

Deploying Docker Containers to Azure Container Instances

- Creates Resource Group as in the example before
- Defines an Azure Container Instances Container Group with an example application
- Uses terraform output.tf to define some output data
- Includes a DNS-Name to make the Application reachable from the Internet

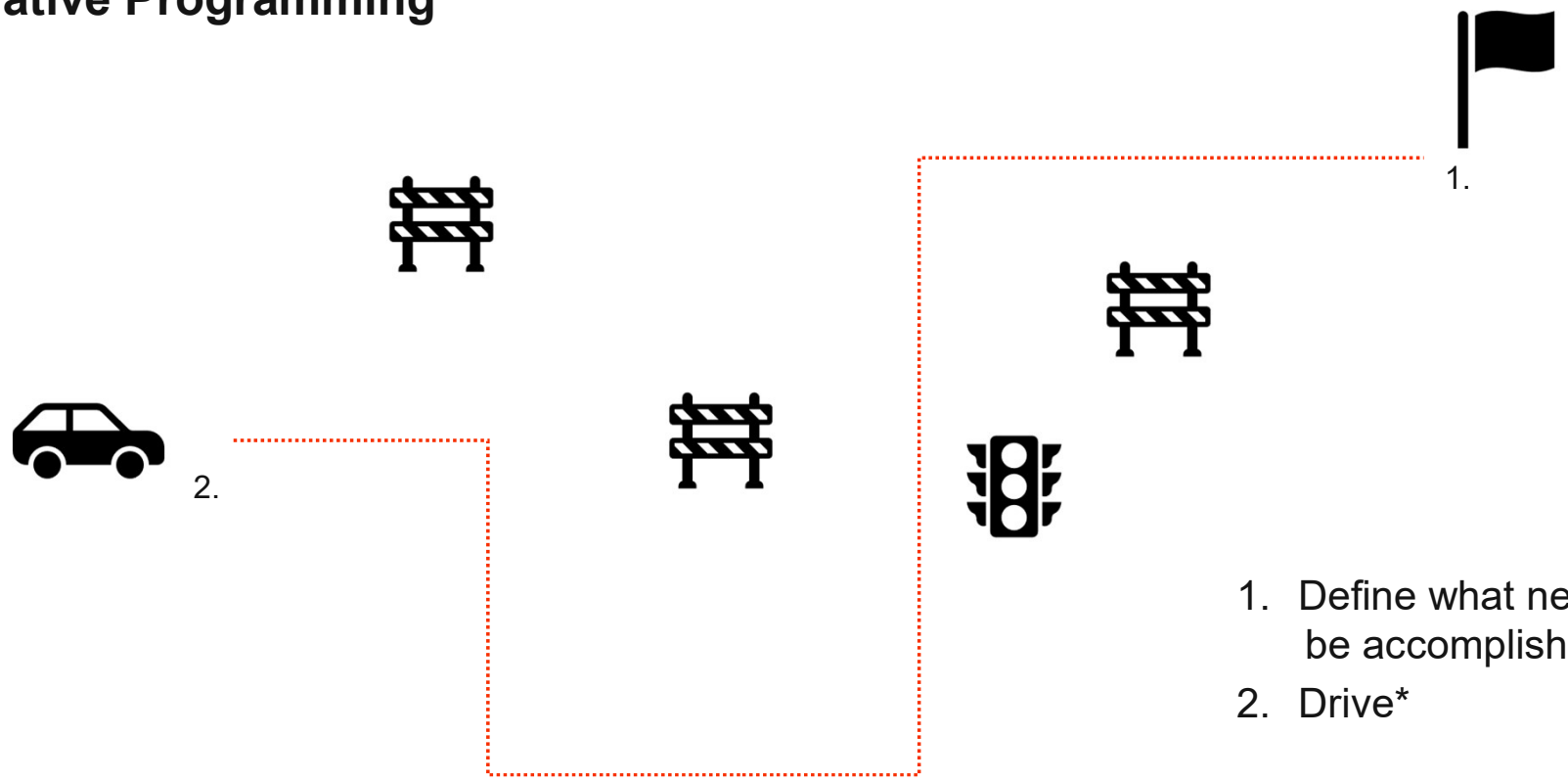
Resource creation in Azure – Terraform dos and don‘ts

- Always exclude the .tfstate files in the .gitignore
- Same applies to the .terraform folder
- Example .gitignore:

```
# Local .terraform directories
**/.terraform/*

# .tfstate files
*.tfstate
*.tfstate.*
```

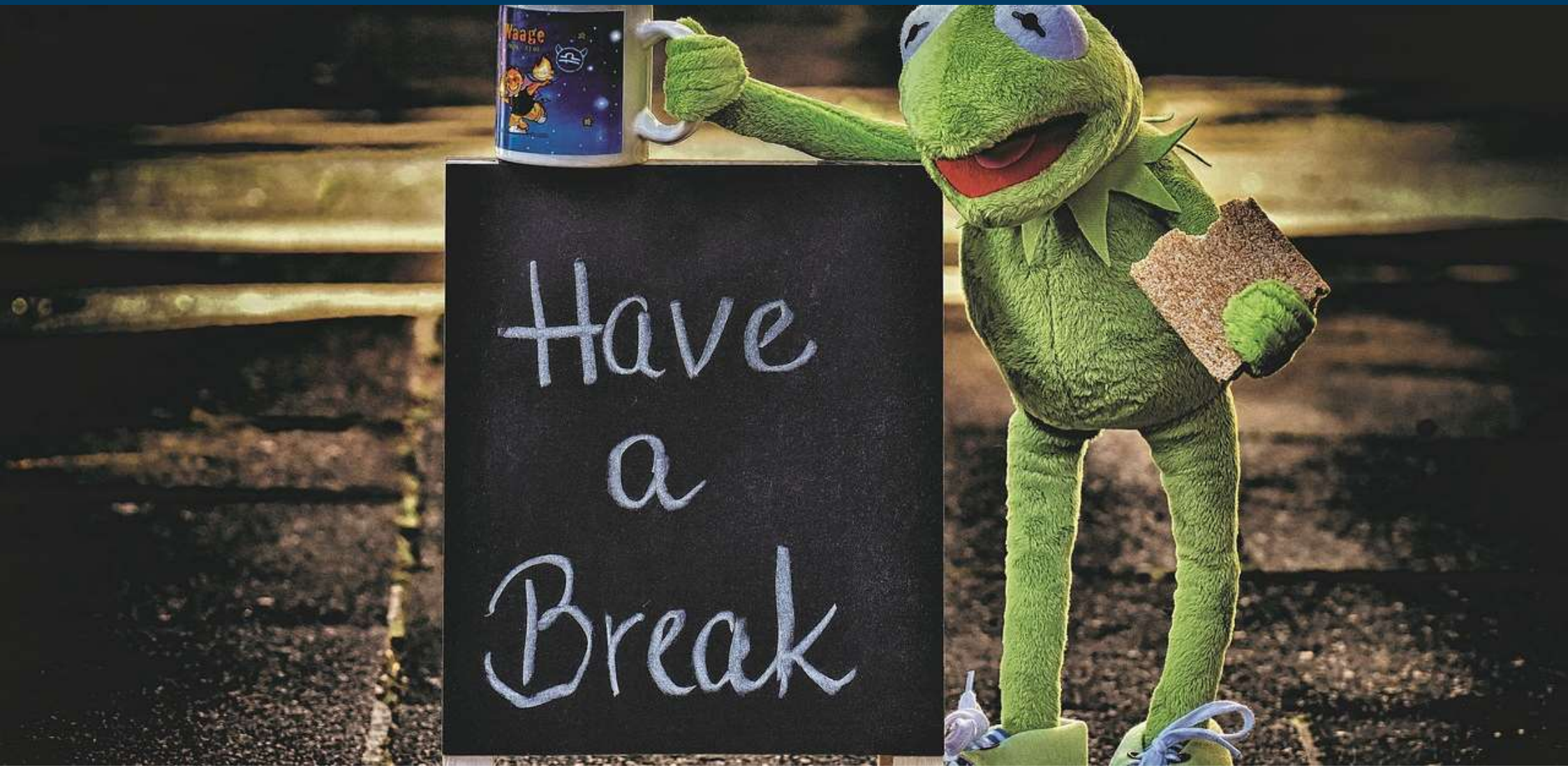
- Don‘t mix different deployment methods in a resource group in Azure
- Use a fixed version of terraform and providers

Declarative Programming



1. Define what needs to be accomplished
2. Drive*

*How to drive is left up to the framework, e.g. Terraform


Waage
Have
a
Break

Collaborative working with Terraform

Collaborative Working – How to share a state

Idea: Use a shared storage to provide the .tfstate file

- If not configured differently, terraform creates its .tfstate file in the current work folder

```
10  provider "azurerm" {
11    skip_provider_registration = true
12    features {}
13  }
14
15  resource "azurerm_resource_group" "resource_group" {
16    name     = "contoso"
17    location = "westeurope"
18  }
19
20  resource "azurerm_storage_account" "storage_account" {
21    name                     = "fgrbssexample2"
22    resource_group_name      = azurerm_resource_group.resource_group.name
23    location                 = azurerm_resource_group.resource_group.location
24    account_tier             = "Standard"
25    account_replication_type = "LRS"
26  }
```

Collaborative Working – How to share a state

```
terraform {
  backend "azurerm" {
    resource_group_name  = "bss-state-file-rg"
    container_name       = "states"
    key                  = "bssworkshop.tfstate"
    storage_account_name = "bssworkshop1tfstates"
  }
}

provider "azurerm" {
  skip_provider_registration = true
  features {}
}

resource "azurerm_resource_group" "resource_group" {
  name     = "contoso"
  location = "westeurope"
}

resource "azurerm_storage_account" "storage_account" {
  name                     = "fgrbssexample2"
  resource_group_name      = azurerm_resource_group.resource_group.name
  location                 = azurerm_resource_group.resource_group.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Idea: Use a shared storage to provide the .tfstate file

- If not configured differently, terraform creates its .tfstate file in the current work folder
- Terraform backend can be configured in your terraform scripts
- This leads to a synchronization between your local deployment and the shared .tfstate file
- Now the infrastructure can be maintained over multiple workstations

Collaborative Working – Where does the storage account come from?

- State storage account must be available before working with remote states – terraform does not create one
- Chicken or the egg dilemma – who handles the state of the state?
- Solution – make it part of the DevOps Pipeline (we will see that in later examples)

Collaborative Working – How to share a state

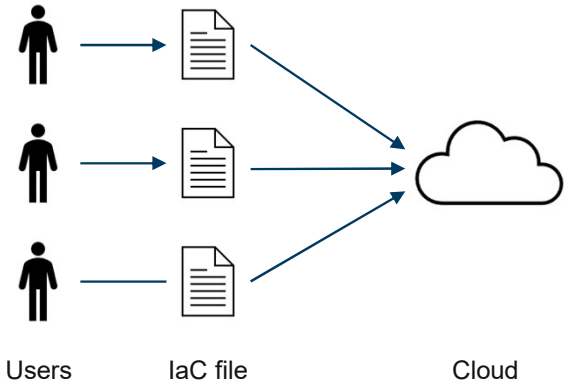
Live Demo

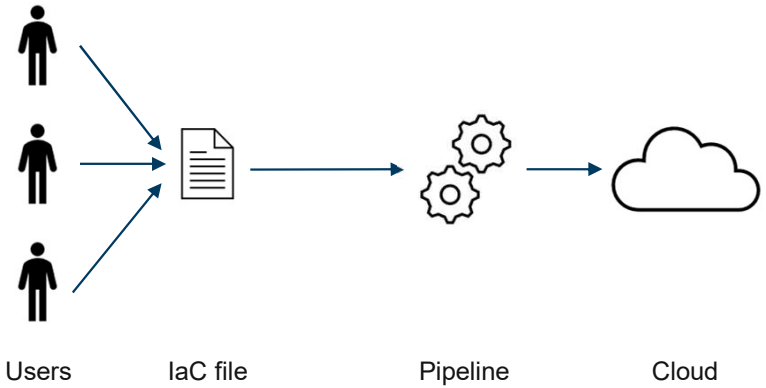
```
terraform {
  backend "azurerm" {
    resource_group_name  = "bss-state-file-rg"
    container_name       = "states"
    key                  = "bssworkshop.tfstate"
    storage_account_name = "bssworkshop1tfstates"
  }
}

provider "azurerm" {
  skip_provider_registration = true
  features {}
}

resource "azurerm_resource_group" "resource_group" {
  name     = "contoso"
  location = "westeurope"
}

resource "azurerm_storage_account" "storage_account" {
  name                     = "fgrbssexample2"
  resource_group_name      = azurerm_resource_group.resource_group.name
  location                 = azurerm_resource_group.resource_group.location
  account_tier             = "Standard"
  account_replication_type = "LRS"
}
```

Idea: Use a shared storage to provide the .tfstate file

- If not configured differently, terraform creates its .tfstate file in the current work folder
- Terraform backend can be configured in your terraform scripts
- This leads to a synchronization between your local deployment and the shared .tfstate file
- Now the infrastructure can be maintained over multiple workstations

Collaborative Working



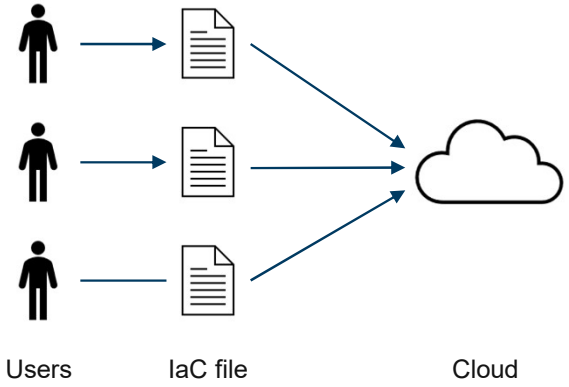


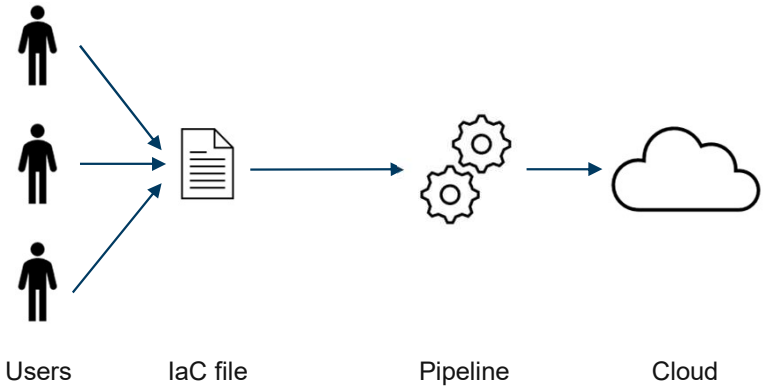
Changes can now be applied from all work stations

- Changes can now be applied from all workstations that have access to the shared .tfstate file
- But: There is no review process, no “good” reusability concept and no approval configuration
- All users that should deploy something to the cloud need the same required permission set
- This could lead to security flaws due to an over permissive role assignment

- **Better: Use continuous integration (GitLab, Jenkins, Azure DevOps etc.)**

Collaborative Working

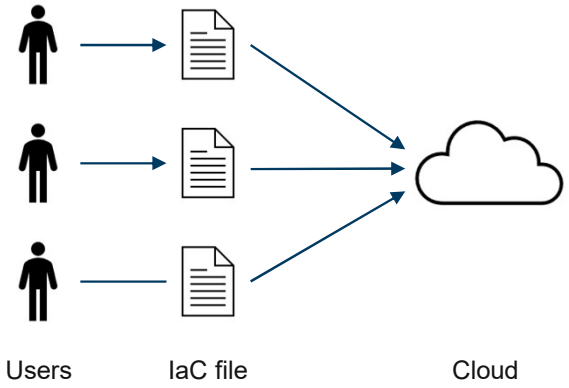


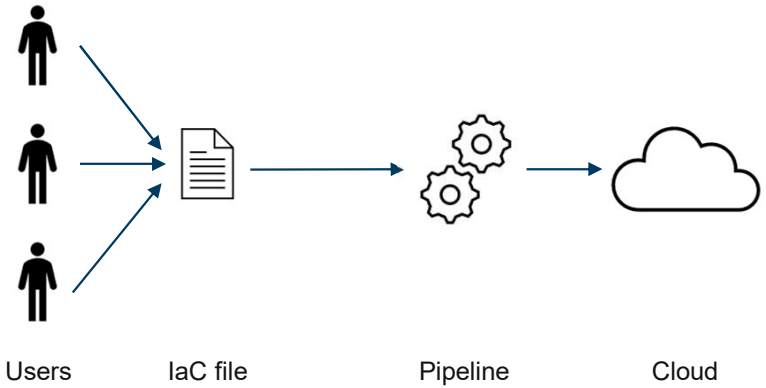


CI / CD in Azure DevOps

- Azure provides a SaaS solution for code development called **Azure DevOps (AzDo)**
- AzDo provides git repositories to share code and pipeline engines, that run on certain agents to execute build and deployment code
- AzDo comes up with additional, useful features such as Azure Active Directory integration, Service Connections, Self hosted agents, approval configurations, deep integration in multiple remote systems (e.g. Azure Kubernetes, Azure Container Registry, Azure Resource Manager, AWS etc.)
- **You can manage your build process along with your code**

Collaborative Working

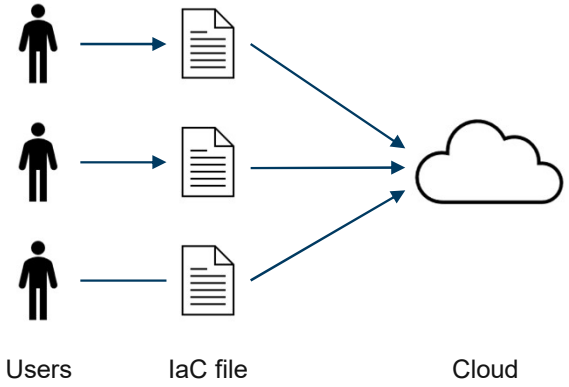


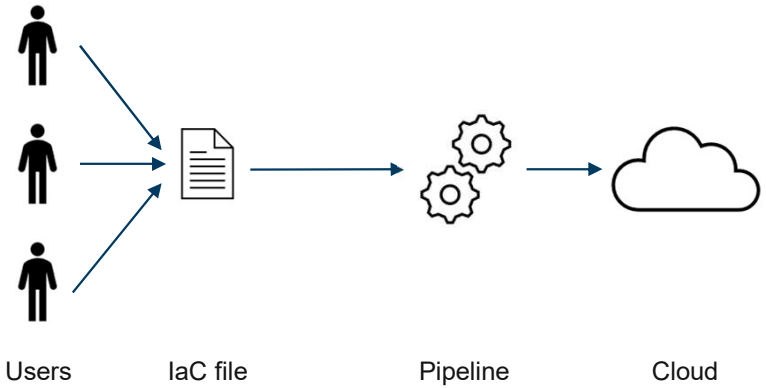


CI / CD in Azure DevOps

- Azure DevOps Pipelines ensure the exact conditions for the development process
- This facilitates reproducibility of builds and deployments and improves the quality of your builds
- With Service Connections, the valid way of deploying solutions can be narrowed down which strengthens the IAM configuration of your cloud resources
  - Not all developers in a project require Contributor (or higher) access to the productive environment
  - There is a way of reviewing and approving committed changes

Collaborative Working



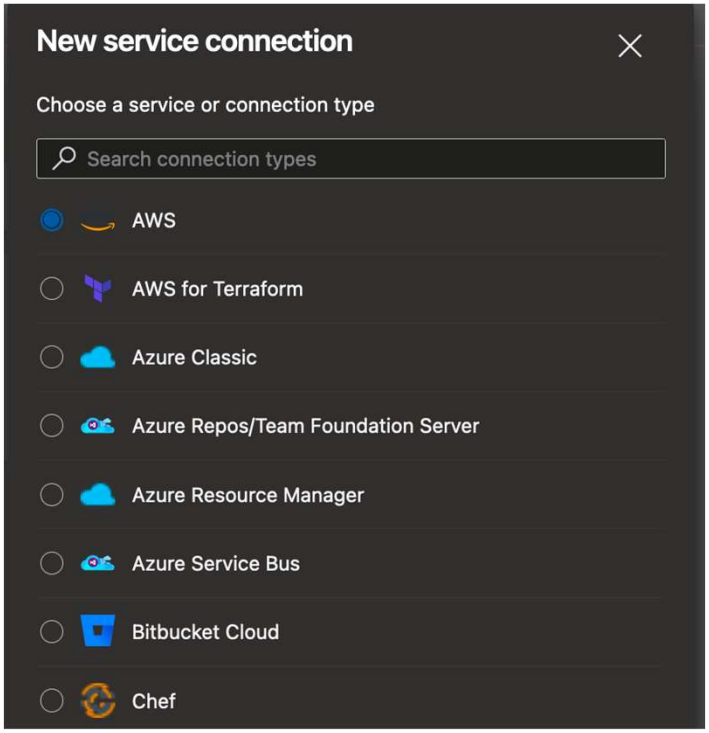


Live Demo

CI / CD in Azure DevOps

- Azure DevOps Pipelines ensure the exact conditions for the development process
- This facilitates reproducibility of builds and deployments and improves the quality of your builds
- With Service Connections, the valid way of deploying solutions can be narrowed down which strengthens the IAM configuration of your cloud resources
  - Not all developers in a project require Contributor (or higher) access to the productive environment
  - There is a way of reviewing and approving committed changes

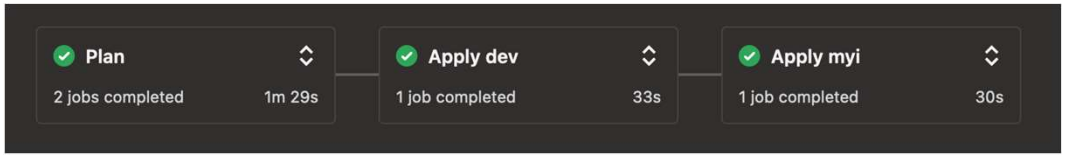
Collaborative Working – Azure Pipelines



Service Connections

- A Service Connection is an abstraction of a Login process
- Service Connections work with different remote systems and facilitate integration of Azure Pipelines
- For Azure, the Service Connection type is called **Azure Resource Manager** and is based on a Service Principal in the target subscription

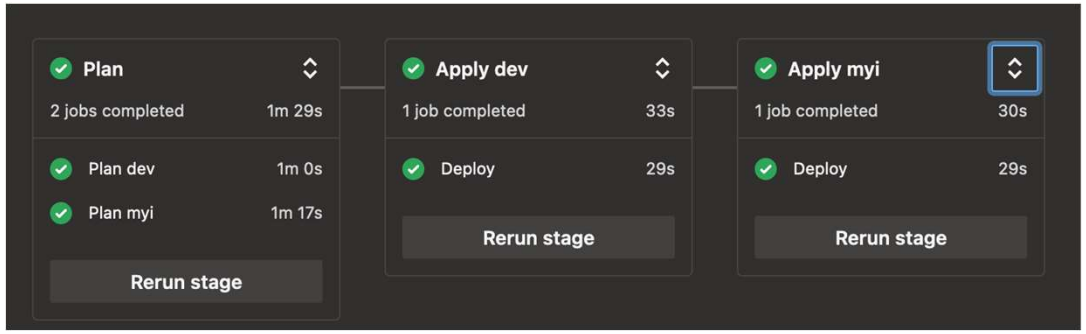
Collaborative Working – Azure Pipelines



CI / CD in Azure DevOps

- There are different ways to deal with automated pipelines in Azure DevOps
  - ”Releases” – a UI based way to form a release pipeline
  - **Multistage Pipelines** – Have your pipeline definition next to your code
- Pipelines are triggered by certain events
  - Timer Trigger
  - Commit Trigger
  - etc

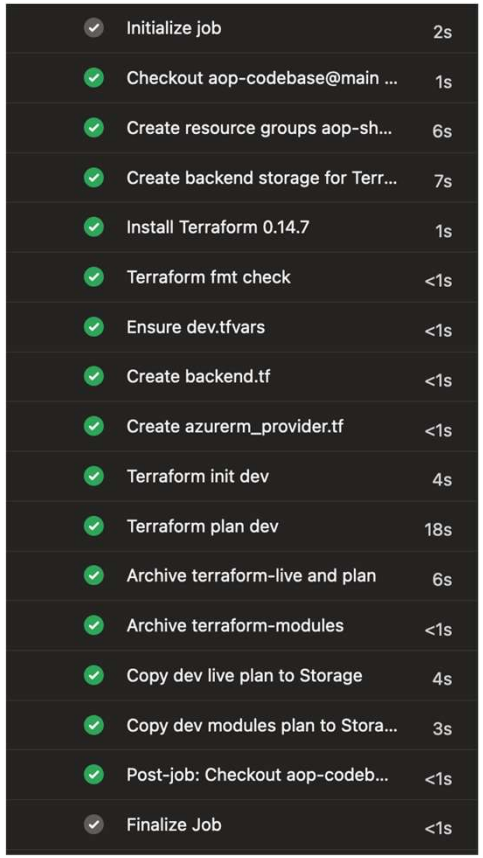
Collaborative Working – Azure Pipelines



CI / CD in Azure DevOps

- An Azure Pipeline consists of
  - **Steps** – The actual tasks to be executed such as Powershell / Bash commands, Uploading tasks, Terraform init / plan / apply etc
  - **Jobs** – Summarize multiple tasks. Multiple jobs can be executed simultaneously. Jobs can be linked to *environments* in Azure DevOps and can be used for Deployment operations
  - **Stages** – Summarize multiple jobs. A stage is used to shape the logical order of your build pipeline. Possible stages are “Build”, “Test”, ”Deploy”

Collaborative Working – Azure Pipelines



Tasks

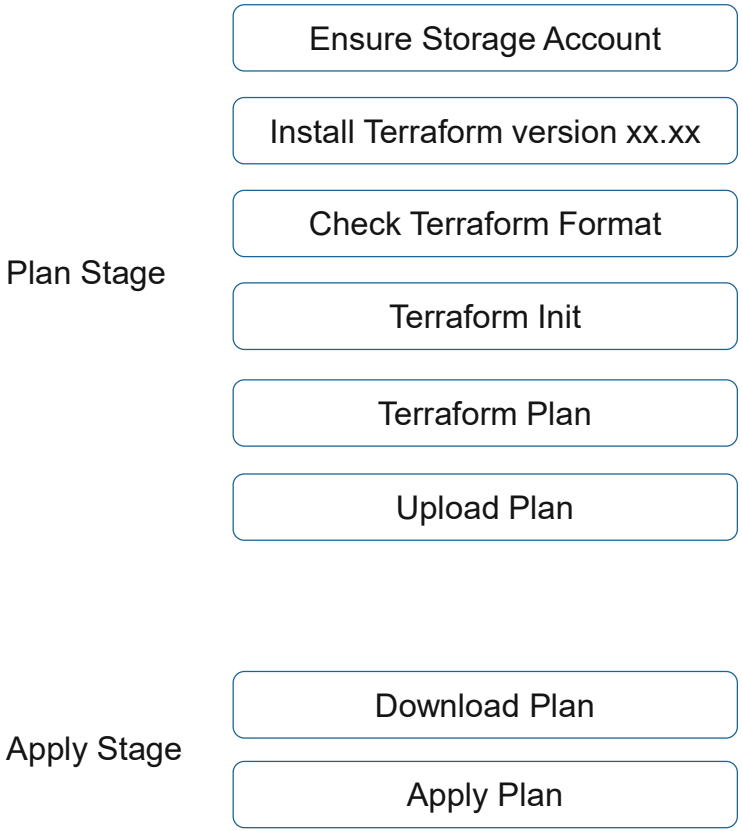
- A task is a single step to be executed
- Tasks work consecutively and in the same working environment
- Specific tasks for engines like Terraform or Powershell can be installed in DevOps using Marketplace feature
- Tasks define a set of valid input and what kind of *service connections* they use
- This highly facilitates integration into remote systems

Collaborative Working – Azure Pipelines

A Terraform Deployment Pipeline

- A good way to think about a pipeline is clarifying on the prerequisites first
  - What Terraform version should be used
  - What checks should be executed on beforehand (such as format checking)
  - Where to store the .tfstate file
  - Are there resources to be built on beforehand?

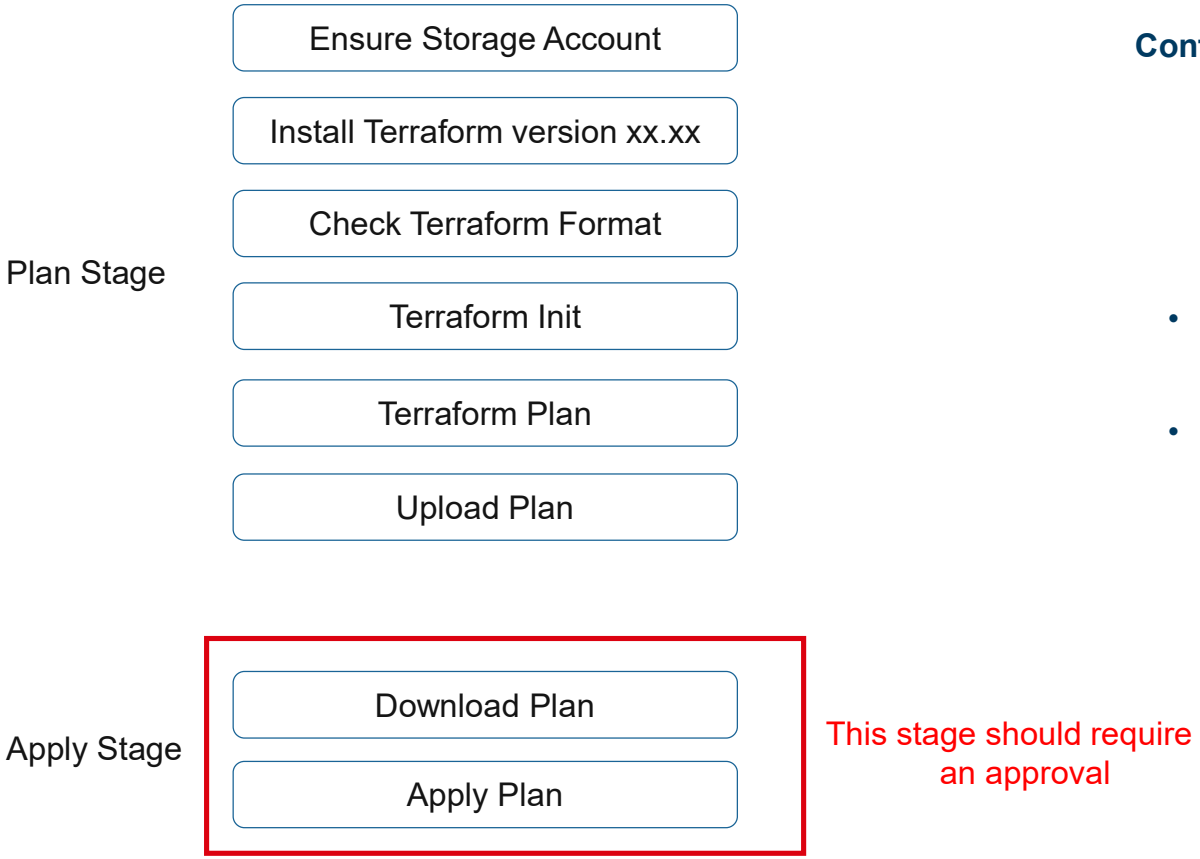
Collaborative Working – Azure Pipelines



A Terraform Deployment Pipeline

- A good way to think about a pipeline is clarifying on the prerequisites first
  - What Terraform version should be used
  - What checks should be executed on beforehand (such as format checking)
  - Where to store the .tfstate file
  - Are there resources to be built on beforehand?

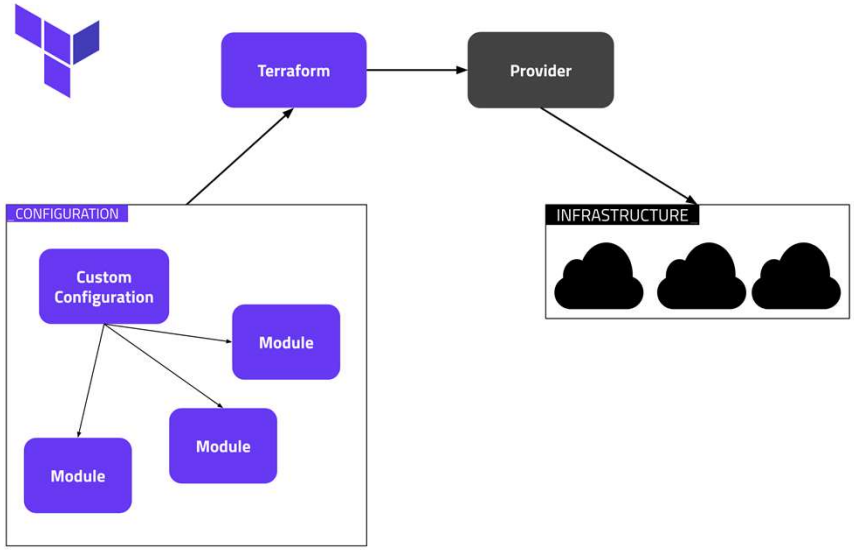
Collaborative Working – Azure Pipelines



Configuring an environment to a stage / job

- Environments are abstract entities that allow you to add approval configuration or monitoring aspects
- Environments can be configured in Azure DevOps under the environment tab in Pipelines Menu

Collaborative Working – Terraform Modules

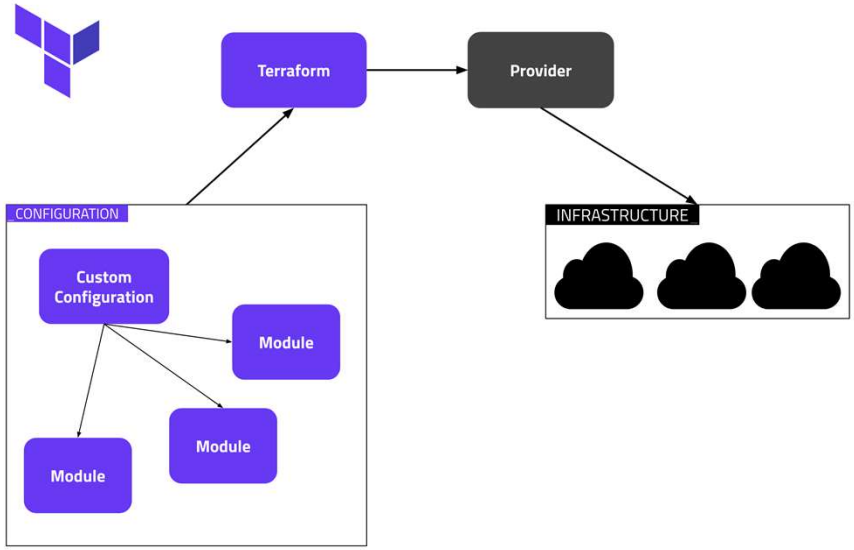


Terraform Modules can group resources that are managed as a block

- Modules are defining input parameters (variables)
- Modules have specified output parameters to work with
- They can be provided though different channels
  - In a git directory
  - Terraform registry
  - Locally
- Modules should not be wrapper for single components, instead they should add a benefit to your terraform scripts

Collaborative Working – Terraform Modules

Live Demo



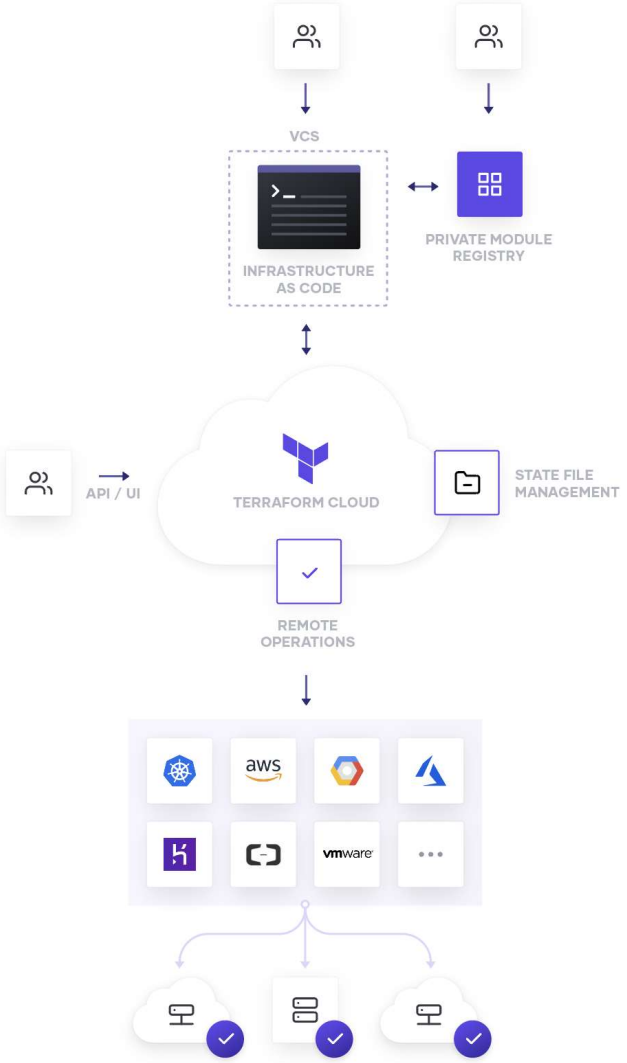
Terraform Modules can group resources that are managed as a block

- Modules are defining input parameters (variables)
- Modules have specified output parameters to work with
- They can be provided though different channels
  - In a git directory
  - Terraform registry
  - Locally
- Modules should not be wrapper for single components, instead they should add a benefit to your terraform scripts

direktgruppe

Collaborative Working – Terraform Cloud

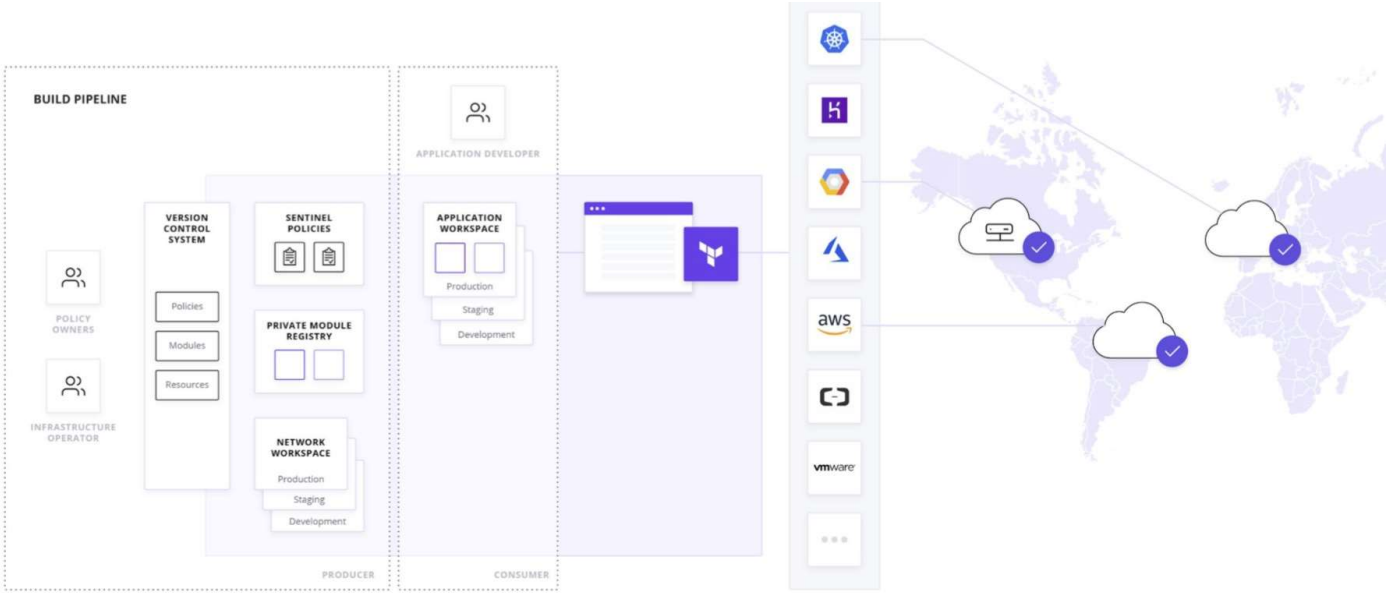
- SaaS solution from HashiCorp
- State files are managed and versioned in Terraform Cloud
- No handling of state storage locations
- IAM tools on the state to manage access
- Terraform can still be used locally for development
- Same workflow with plan, apply stages
- Hides and automates Terraform runs and state handling
- Private module registry
- Optional:
  - Terraform talks to and runs in Terraform Cloud
  - Versioning of runs
  - Approval checks in the workflow



direktgruppe

Collaborative Working – Terraform Enterprise

- Self-hosted solution
- Targets organizations who have special requirements regarding localization and data protection
- Includes operational policies
- Same benefits as with Terraform Cloud



direktgruppe

Collaborative Working – Terraform State operations

```
1   # list all resources in the current state
2   terraform state list
3
4   # import an existing resource with an Azure ID into
    your state for a certain mapping (ADDR)
5   terraform import ADDR ID
6
7   # Remove a resource with the identifier ADDR out of
    your state
8   terraform rm ADDR
```

The terraform command has certain options to

- The terraform command can configure the statefile
- Operations are e.g.
  - `terraform import [options] ADDR ID`
  - `terraform state list`
  - `terraform state rm ADDR`

Collaborative Working – Why Terraform? What benefits compared to older methods

Advantages

- Most popular tool for Multi-Cloud scenarios
- Easy to learn syntax
- Better readability as e.g. ARM-Templates
- Remote state for collaborative working
- Integrates with CI/CD Tools
- Providers for many different clouds and technologies
  - All major public clouds available: Azure, AWS, GCP
- One syntax for different infrastructure code
  - Replaces ARM Templates, CloudFormation etc.

Disadvantages

- Terraform providers are always a little behind the overall development of cloud providers
- There is still a need for some „glue scripts“ etc.
- Handling the state of the state (chicken or egg dilemma)
- New tool and processes must be established
- IAC code “rottens” quite fast
- Code must be maintained after major version updates to terraform

Feedback & QnA

https://forms.office.com/r/xLRiHHKQ8H

Vielen Dank für
Ihre Aufmerksamkeit!

direkt gruppe
Hamburg I Griegstraße 75 I Haus 26 I 22763 Hamburg
Köln I Im Mediapark 6b I 50670 Köln
München I Landaubogen 1 I 81373 München
Paderborn I Technologiepark 9 I 33100 Paderborn
Tel. +49 40 88155-0 I info@direkt-gruppe.de
www.direkt-gruppe.de I blog.direkt-gruppe.de

direktgruppe