# AKS Complex Solutions CaseStudy1

Last updated by | Andreas Walter | Mar 27, 2021 at 7:19 AM GMT+1

## AKS Complex Solutions Case Study 1

Contents



### Challenge

Our Customer has an existing Application built on Azure IaaS and PaaS solutions. All systems had been setup using Terraform as IaC tool. Starting on the basis of this code, our customer asked us to check the environment against Azure Best Practices, optimize the code and the pipelining, while taking into account scalability and release management as well as security. The new environment is to be setup in a new release including the optimizations to pipelining. All resources are to be managed by our managed service department to take care of monitoring and availability and troubleshoot issues, should they arise.

### Microsoft Technologies

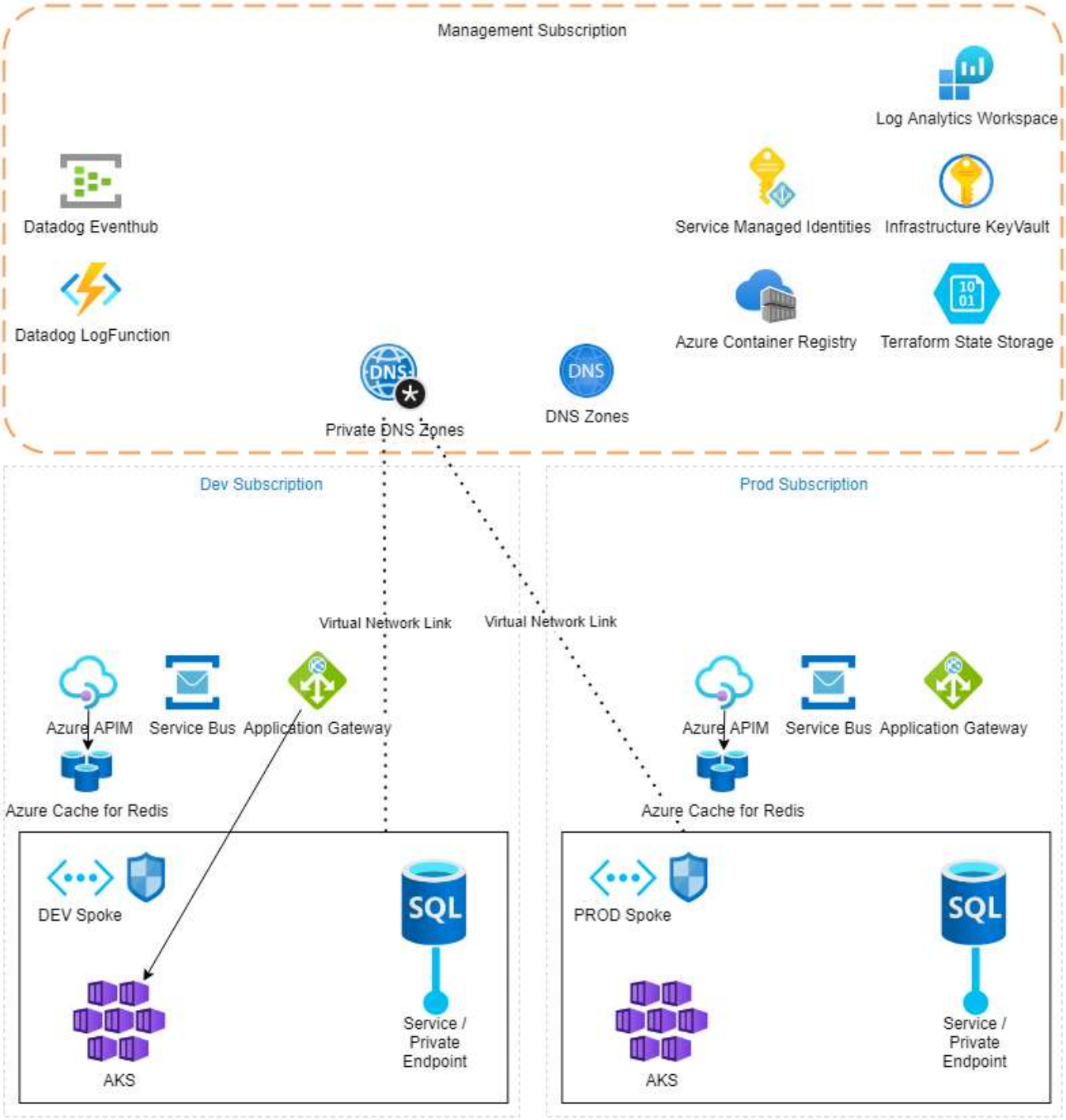
- API Management Gateway
- Application Gateway
- Azure Cache for Redis
- Azure DNS / Private DNS Zones
- Azure Functions
- Azure Kubernetes Services
- Container Registry
- Event Hub
- Key Vault
- Log Analytics Workspace
- Private Endpoint
- Service Bus

- SQL Databases
- Storage Accounts
- Virtual Network

### Target App-Infrastructure

The customers application consists of the basic elements, AKS Cluster, Service Bus, API Management Gateway and Application Gateway. Communication should be secured where possible by using vnet traffic only, while keeping cost to a minimum. The application entry point is the Application Gateway. The application gateway is the applications entrypoint and distributes traffic to the pods. The Application Gateway also provides the WAFv2. For a rough description of what is every service used for take a look at the following table:

| Service | Usage |
| --- | --- |
| API Management Gateway | Gateway for all mobile devices to Azure resources |
| Application Gateway | Webapplication Firewall v2 and entrypoint to AKS |
| Azure Cache for Redis | Redis Cache for API Management |
| Azure DNS / Private DNS Zones | Dynamic DNS for all services and private endpoint DNS settings |
| Azure Functions | Datadog log forwarder for monitoring |
| Azure Kubernetes Services | Main application services are hosted in Kubernetes |
| Container Registry | Contains all the customers own container images |
| Event Hub | Application event dispatching / Datadog Logforwarding |
| Key Vault | Secures all of the environments secrets and certificates |
| Log Analytics Workspace | Captures logs that do not get streamed to datadog |
| Private Endpoint | Secures the traffic for SQL Databases |
| Service Bus | Application message dispatching |
| SQL Databases | Application database, one database per application |
| Storage Accounts | Terraform state storage and possible application persistent storage |
| Virtual Network | Provides communication to all "Close-to-IaaS" services |




## Migration to new environment

The main code of the infrastructure is a rewrite of the previous environment. All existing Terraform modules were rewritten, to observe security and application best practices, where possible. Conceptually the Networking infrastructure was setup first.

The old infrastructure used wider network address spaces than currently permitted by the customers network department for Azure environments. Since in the future it might become necessary to add a hub network and peering, all IP-address ranges had to be coordinated with the customers network department. Therefore the maximum amount of addresses required by the application was discussed with the customer.

One environments infrastructure (either TEST or PROD) consists of one `/21` that is divided in several subnets.

| Environment | VNET address space | Cluster subnet | WAF subnet | Private Endpoint subnet |
|---|---|---|---|---|
| Prod | 10.123.0.0/21 | 10.123.0.0/22 | 10.123.4.0/24 | 10.123.5.0/26 |
| Test | 10.123.8.0/21 | 10.123.8.0/22 | 10.123.12.0/24 | 10.123.13.0/26 |

Previously the vnet creation was included in the Terraform module for the AKS cluster, since the VNET was not only used for just AKS it was decided to move the configuration from the AKS Cluster to a separate VNET module. As can be seen in the following code, the creation of the network includes the virtual network links to the private DNS zones and is dependent on the management module being deployed beforehand.

Reworked VNET module (Codesnippet)

```
resource "azurerm_virtual_network" "serviceVnet" {
  address_space       = [var.clusterVnetAddressSpace]
  location            = var.location
  name                = format("%s-vnet", var.resourceNamePrefix)
  resource_group_name = var.resourceGroupName

  tags = var.tags
}

resource "azurerm_private_dns_zone_virtual_network_link" "pdnsVnetLink" {
  for_each              = toset(var.privateLinkZones)
  name                  = replace("${each.value}-${azurerm_virtual_network.serviceVnet.name}",".","-")
  resource_group_name   = format("%s-rg", local.resourceGroupMgmtNamePrefix)
  private_dns_zone_name = each.value
  virtual_network_id    = azurerm_virtual_network.serviceVnet.id
  provider = azurerm.mgmt
}

resource "azurerm_subnet" "peSubnet" {
  address_prefixes     = [var.privateEndpointsAddressSpace]
  name                 = format("%s-subnet-pe", var.resourceNamePrefix)
  resource_group_name  = var.resourceGroupName
  virtual_network_name = azurerm_virtual_network.serviceVnet.name
  service_endpoints = var.subnetServiceEndpoints
  enforce_private_link_endpoint_network_policies = true
}

resource "azurerm_subnet" "k8sSubnetCluster" {
  address_prefixes     = [var.clusterPodSubnetAddressSpace]
  name                 = format("%s-subnet-cluster", var.resourceNamePrefix)
  resource_group_name  = var.resourceGroupName
  virtual_network_name = azurerm_virtual_network.serviceVnet.name
  service_endpoints = var.subnetServiceEndpoints
}

resource "azurerm_subnet" "k8sSubnetWaf" {
  address_prefixes     = [var.clusterServiceSubnetAddressSpace]
  name                 = format("%s-subnet-waf", var.resourceNamePrefix)
  resource_group_name  = var.resourceGroupName
  virtual_network_name = azurerm_virtual_network.serviceVnet.name
}
```

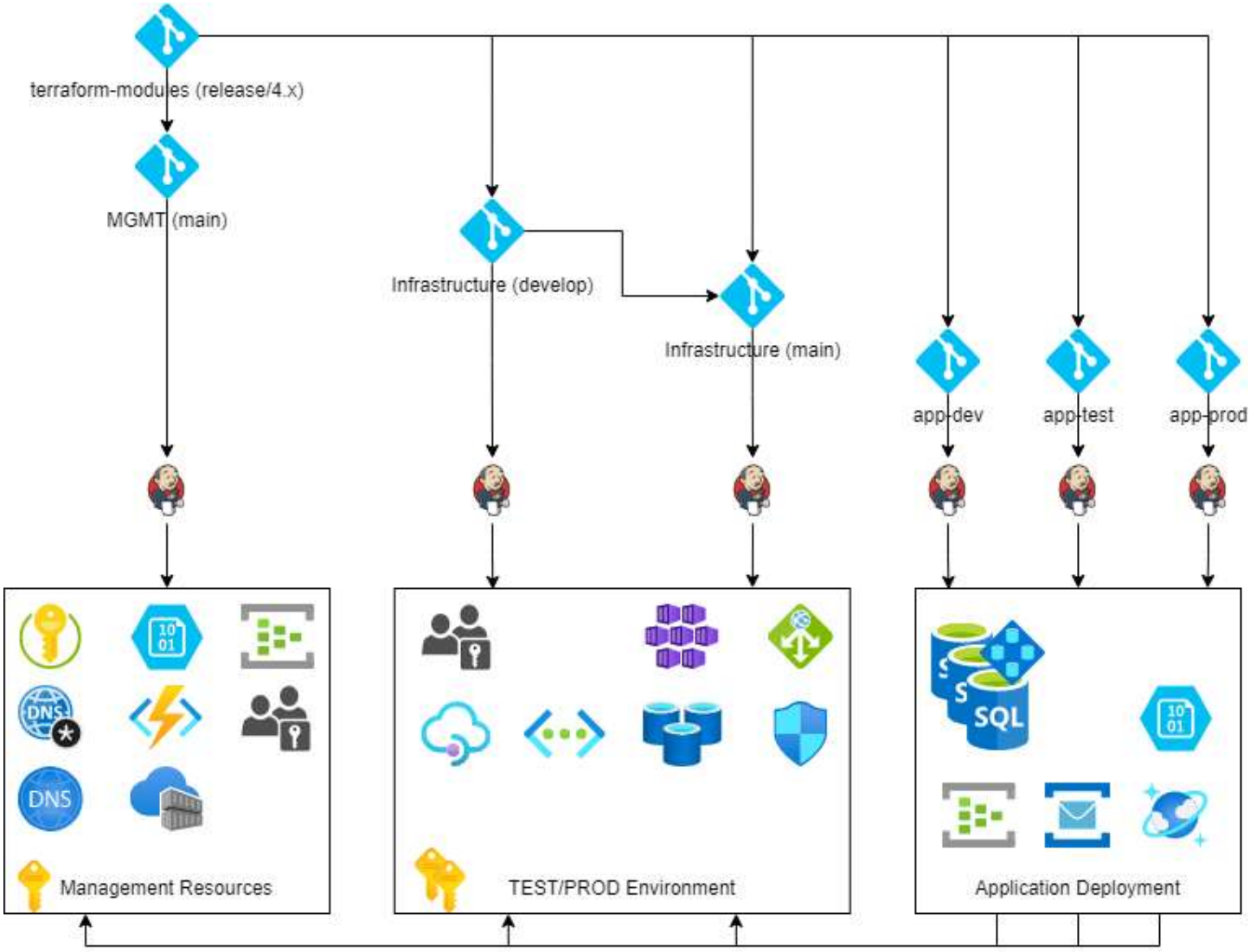
The following graphic should visualize how repositories are dependent from one another. The first to be deployed is the management infrastructure. If this is a first time deployment, the Terraform state storage does not exist yet. Therefore it is necessary to do a local bootstrap of the management infrastructure. After the state storage is created, the state is migrated to the state storage and management infrastructure can then be deployed using pipelines. After the management infrastructure is setup, all necessary administrative role assignments are done and the management subscription is to be considered onboarded.

The app infrastructure is split into two subscriptions, where the test and dev application share the infrastructure of the TEST-subscription and the production deployment runs on the Azure PROD-subscription the app infrastructure deployments also do a basic setup of the AKS clusters, to deploy some management namespaces and helm charts, such as kured or ingress and egress resources.

Deployment of the kured helm chart (Codesnippet)

```
provider "helm" {
  kubernetes {
    host     = module.k8sCluster.k8sConfig.host

    client_key             = base64decode(module.k8sCluster.k8sConfig.client_key)
    client_certificate     = base64decode(module.k8sCluster.k8sConfig.client_certificate)
    cluster_ca_certificate = base64decode(module.k8sCluster.k8sConfig.cluster_ca_certificate)
  }
}

resource "helm_release" "kured" {
  chart = "kured"
  name = "kured"
  repository = "https://weaveworks.github.io/kured"
  namespace = local.namespaceName

  set {
    name = "image.tag"
    value = var.kuredVersion
  }
  set {
    name = "configuration.timeZone"
    value = var.timeZone
  }
  set {
    name = "configuration.startTime"
    value = var.rebootStartTime
  }
  set {
    name = "configuration.endTime"
    value = var.rebootEndTime
  }
  set {
    name = "maxUnavailable"
    value = var.maxNodesUnavailable
  }
  values = [
    <<EOF
    configuration:
      rebootDays: ${var.rebootDays}
    EOF
  ]

  depends_on = [module.namespace]
}
```

In the third step, the application specific repositories are deployed. These repositories target all subscriptions, since in the management subscription there might be some shared infrastructure such as DNS Zones that need to be populated.




## Configuration of plattform monitoring

Our standard monitoring currently is Datadog, therefore Datadog was deployed using Helm charts on the AKS cluster. Monitoring of Azure Resources is resolved using the a Datadog Service Principal with the "Monitoring Reader" permissions and Datadog handles Subscription and Resource discovery.

For Azure Activity Logs this process isn't that easy, Datadog requires an Eventhub and a Function App that pushes the diagnostic information and logs to the Datadog-Service. Datadog does only provide ARM-Onboarding for these resources, however a requirement for us was to not create secondary deployments. Therefore the ARM-deployments were converted to Terraform.

As is always with Terraform some features are not available in Terraform. One example is the deployment of the actual function itself. However due to Terraform being able to deploy ARM-templates into an environment this issue can be easily resolved.

> Note: that the following example uses an `azurerm_template_deployment` resource rather than the new `azurerm_resource_group_template_deployment` this is due to the fact, that the new resource has a bug at time of writing this casestudy (azurerm provider version 2.52 and earlier fixed in 2.53).

Datadog function deployment (ARM-Workaround) (Codesnippet)

```
resource "azurerm_template_deployment" "functionDatadog" {
  name = "${azurerm_function_app.functionAppDatadog.name}-datadog-log-forwarder"
  resource_group_name = var.resourceGroupName
  deployment_mode = "Incremental"
  parameters = {
      "functionCode" = file("${path.module}/index.js")
    }
  template_body = <<TEMPLATE
{
    "$schema": "https://schema.management.azure.com/schemas/2019-04-01/deploymentTemplate.json#",
    "contentVersion": "1.0.0.0",
    "parameters": {
        "functionCode": {
            "type": "String",
            "metadata": {
                "description": "Code for the function to run, saved into index.js"
            }
        }
    },
    "variables": {},
    "resources": [
        {
            "type": "Microsoft.Web/sites/functions",
            "apiVersion": "2020-06-01",
            "name": "${azurerm_function_app.functionAppDatadog.name}/datadog-log-forwarder",
            "dependsOn": [],
            "properties": {
                "config": {
                    "bindings": [
                        {
                            "name": "eventHubMessages",
                            "type": "eventHubTrigger",
                            "direction": "in",
                            "eventHubName": "${azurerm_eventhub.eventHub.name}",
                            "connection": "Datadog-EventHub-AccessKey",
                            "cardinality": "many",
                            "dataType": "",
                            "consumerGroup": "$Default"
                        }
                    ],
                    "disabled": false
                },
                "files": {
                    "index.js": "[parameters('functionCode')]"
                }
            }
        }
    ]
}
TEMPLATE
}
```

## Permissions Using Managed Identities

One of the best practices we adhered to was to minimize the usage of fixed credentials in the deployments. So every service was configured to use either `UserAssigned` or `SystemAssigned` managed service identities. Since the Terraform service principals are dedicated to each subscription it is necessary to add `Owner` / `User Account Admninistrator` permissions to some resources in the management subscriptions. One example is the ability to assign `AcrPull` permissions to the AKS clusters MSI.

In the ACR module the Terraform service principals were added as Owner to the container registry

ACR RBAC assignment(Codesnippet)

```
resource "azurerm_role_assignment" "RegistryOwner" {
  for_each = toset(var.workloadSubSPObjectIds)
  scope                = azurerm_container_registry.containerRegistry.id
  role_definition_name = "Owner"
  principal_id         = each.key
}
```

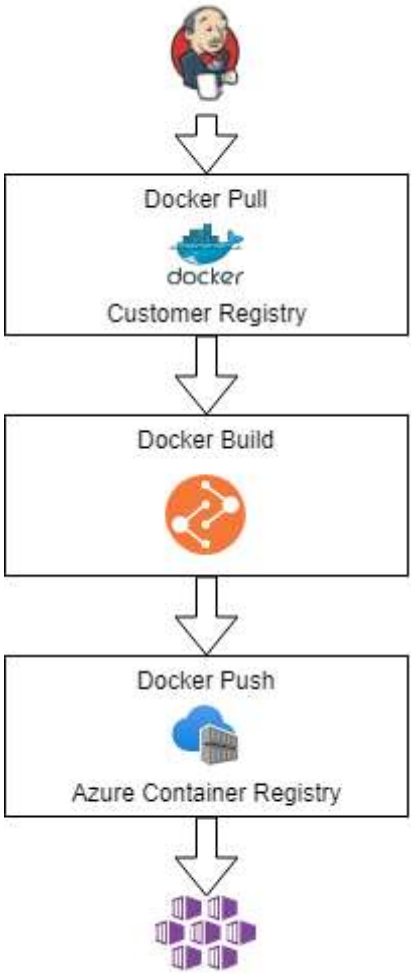
In the K8s Module the Kubelet identity was then added to the `AcrPull` role to enable a connection to the shared container registry.

ACR Pull rights (Codesnippet)

```
resource "azurerm_role_assignment" "k8sServicePrincipalRightsArcPull" {
  scope                = data.azurerm_container_registry.containerRegistry.id
  role_definition_name = "AcrPull"
  principal_id         = azurerm_kubernetes_cluster.k8sCluster.kubelet_identity.0.object_id
}
```

## Docker Image deployment

The images for the Azure container registry are prepared by building images from the customer on premises repository and pushing the images to the container registry using Jenkins pipelines.



```
FROM repo.customer.com:8020/<repo>/<imageName>:<imageTag>

COPY --chown=jboss deployments $JBOSS_HOME/standalone/deployments/

COPY --chown=jboss property /opt/jboss/property/

RUN touch $JBOSS_HOME/standalone/deployments/<DODEDPLOYFILE>.dodeploy

USER root
RUN curl -k https://<customerCAURL>.customer.com:8443/CAName --output $JBOSS_HOME/CAName.cer \
&& keytool -import -trustcacerts -keystore "$JAVA_HOME/lib/security/cacerts" -alias CAName -storepass chang
&& rm $JBOSS_HOME/CAName.cer

USER jboss
```

## Conclusion

This project enabled our customer to redesign their current infrastructure and enable managed services for Azure IaaS and PaaS resources were necessary. The fully automated infrastructure deployment enables the customer to work with release based upgrades of their infrastructure while still having the ability to customize the environment for their customers based on Terraform modules.