IT Infrastructure & Automation: CA1

Craig Dillon
Technological University Dublin
x00205790@mytudublin.ie

Introduction

Our client is an online retailer who is considering moving their web application from on premise to the public cloud. They have concerns regarding data security, privacy and GDPR as a result of a recent GDPR breach and may decide to retain customer data on premises. There are also issues regarding workflow, testing and general maintenance of their current architecture.

Infrastructure design

I propose moving the client to managed Kubernetes clusters in Azure (AKS). Terraform will deploy a control plane cluster, production, staging and development clusters along with respective hosted mysql databases. Once these are deployed ArgoCD will then be installed on the control plane allowing the developers to use continuous deployment of applications and services to each respective cluster. AKS is deployed with an application gateway ingress controller (AGIC) which allows applications on the AKS cluster to expose itself to the internet via an ingress gateway. The Terraform state is stored in an Azure blob, although GitLab also has a feature for storing statefiles.

This deployment is handled within GitLab and uses a docker runner to deploy the manifests and run some installation commands.

The proposed workflow is as follows;

- A Terraform manifest creates AKS clusters, AGICs, database servers and databases
- Once deployed, ArgoCD is installed on the control plane cluster
- ArgoCD is now available to deploy applications, services not only to the control plane but also to the other AKS clusters and can pull these configurations from repositories

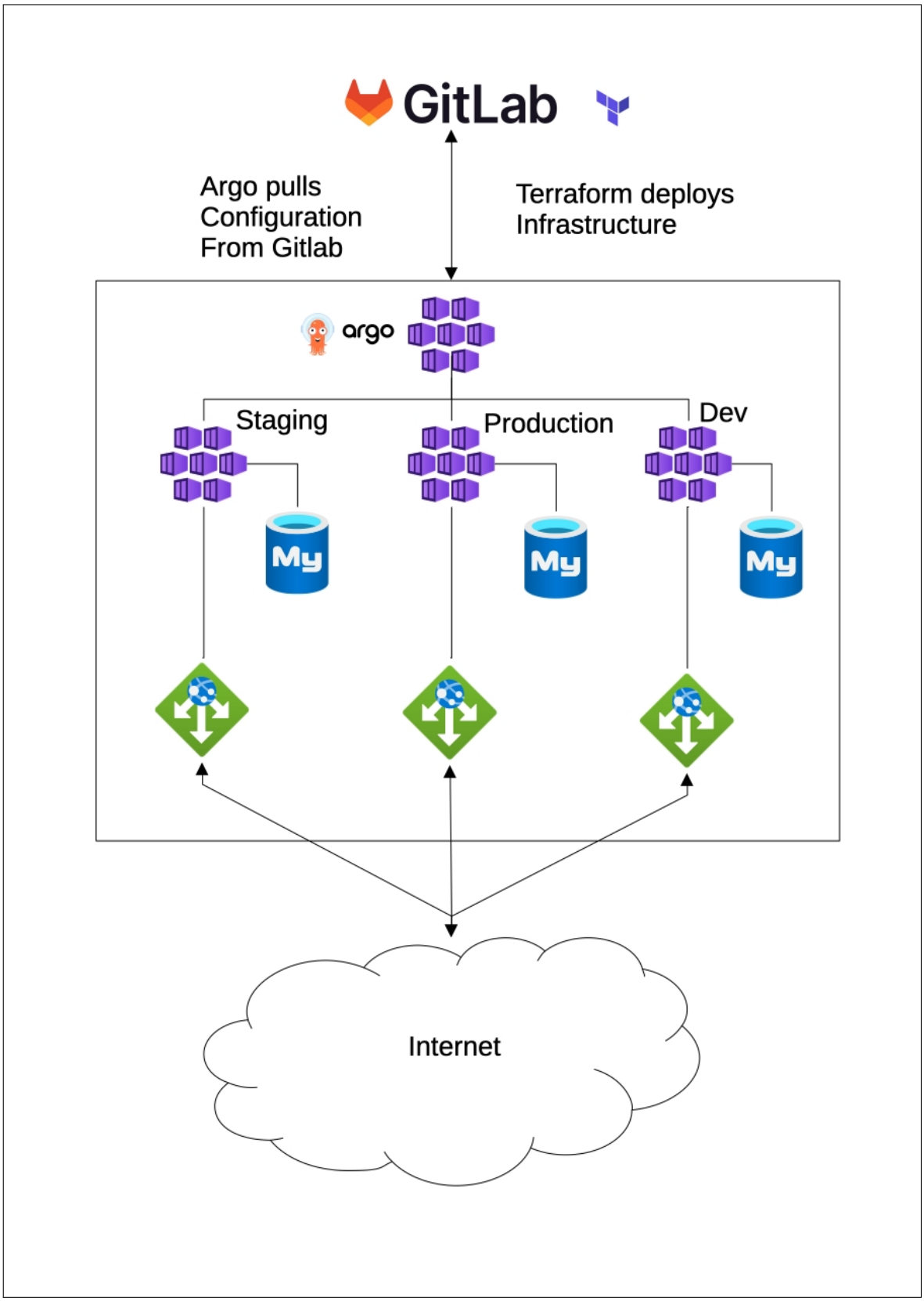

Figure 1: Architecture Diagram

Solution to the highlighted issues

Following this design we tackle the highlighted issues by providing developers with an environment that is managed, the infrastructure deployment method is code which can be redeployed. This removes the problem of continuous maintenance tasks. Each cluster and database is identical, reducing the potential for production errors that do not occur during testing and staging yet show up in production. These clusters are scalable and the Terraform manifests can be modified to increase node pool sizes or even to create more AKS clusters if required.

The AKS clusters are scalable and this can be amended in the manifests also, which should remove any issues of unavailability of test and staging environments. By using the provided Terraform manifests as templates, the developers also have the freedom of creating their own cluster for their own application should they see fit.

This infrastructure is managed by Azure, which means that very few people, if any, should have to interact with the clusters directly after the initial setup. Any changes to the infrastructure can be tested on a non production environment before pushing to the production environment. ArgoCD allows the developers to adopt continuous delivery, that is, ArgoCD will monitor the desired repositories and pull when there are any changes. There are other advantages here too, you can apply a specific version to the pull requests or deploy several versions of an application into the cluster for testing or development.

Using Terraform manifests to deploy ensures stability, consistency and security. According to Alonso et al. (2022) ”With a strict config, it is possible to avoid creating config drifts which again improves the overall security posture of the deployed solution.” It is possible to schedule Terraform to run a plan job regularly, which will compare the state in code to the state in the live environment and a redeployment can be triggered if required.

As you can see, there are relatively few components deployed per cluster which brings cost and performance benefits as well as something that is easily repeatable and scalable. In comparison, building a cluster of VMs or scale sets would also bring an increased requirement on maintenance as the underlying OS still needs to be maintained by the client and if they wish to maintain the state of certain packages then they would need to make use of a configuration management tool like Puppet or Chef for example.

GDPR concerns and recommendations

The security of AKS clusters is handled as part of the Azure managed service. The Kubernetes API server is accessible by default using a public IP address but the endpoint can be limited to specific IP ranges, which I would recommend. There is also the option of creating a fully private AKS cluster on Azure. The AKS nodes themselves are maintained by the customer, however, for Linux based nodes there are nightly updates rolled out by Microsoft. When deployed, the nodes are placed in a private network space and have no connection to the internet. Storage on the nodes is encrypted at rest.

The database is a managed service with a MySQL database running, this is available for connection using TLS for applications on any of the AKS clusters.

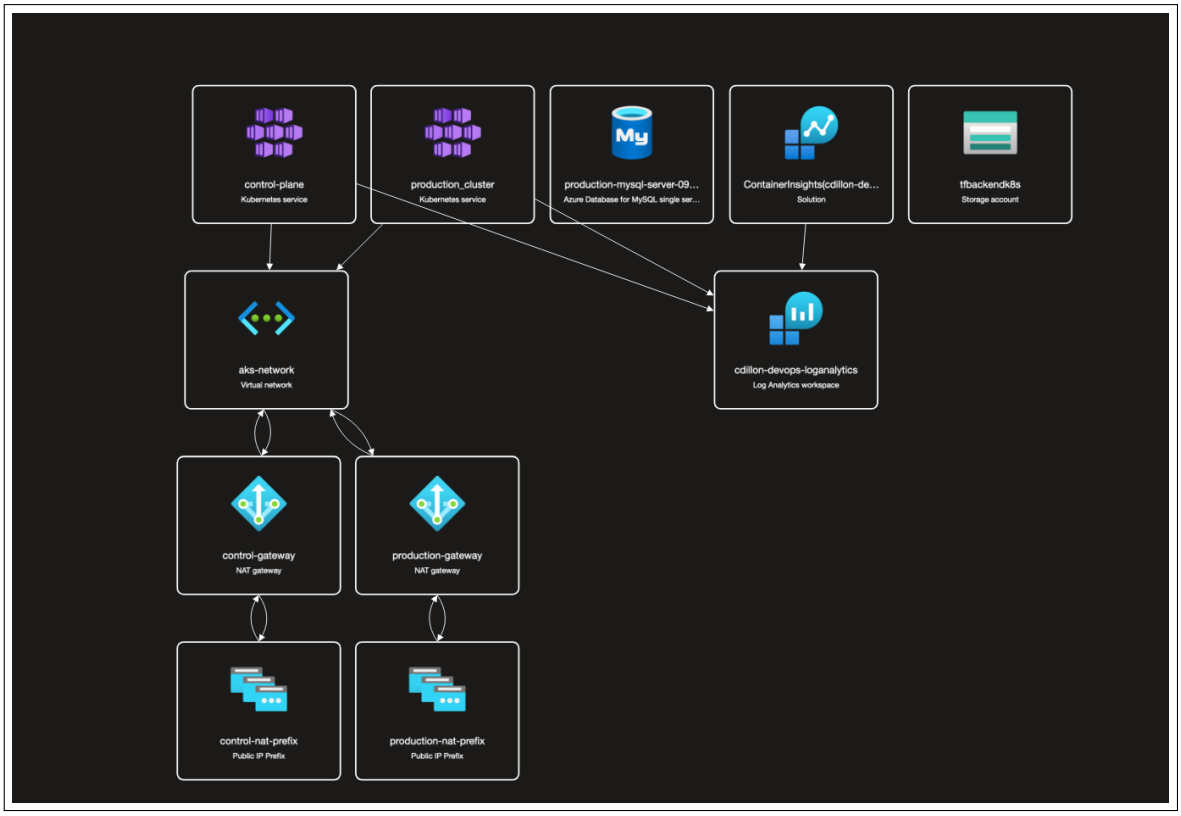

Figure 2: An example of an Azure deployment using resource visualiser

The database service is double encrypted, that is, there are customer managed encryption keys and then infrastructure layer keys applied. The customer managed keys can be either system assigned by Azure or customers can use their own keys if so desired.

If the customer wishes to use a private cloud, for example VMware or HyperV on premises or hosting in a dedicated datacentre then they will need to open connections to allow this application to make use of the data stored there. This, in my opinion, increases risk as that data flow is occurring across the internet. Adding a VPN or other dedicated connection adds a level of security but will also increase the level of complexity. The idea here is to reduce maintenance and potential downtime, having the entire environment self-contained achieves this.

Additional features

ArgoCD is an incredibly powerful tool when using Kubernetes clusters. It allows for management of several clusters, can make use of applications with in the Kubernetes ecosystem and can also be used as a trigger for deploying your own applications. Other benefits of using Kubernetes would be making use of monitoring either natively through the Azure Log Analytics which is deployed in the supplied Terraform manifest, or by using Prometheus or even an Azure managed instance of Prometheus. There are several out of the box solutions for monitoring available that would enable developers to create alert patterns and act before resources are exhausted.

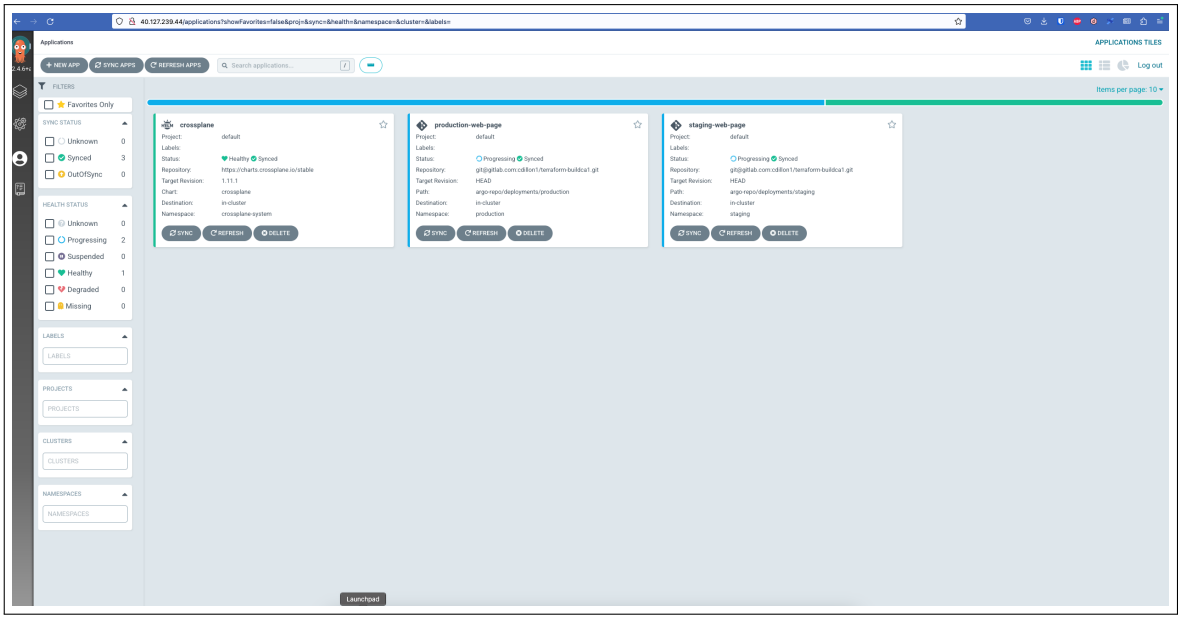

Figure 3: ArgoCD dashboard

Using Kubernetes as the core platform means that the client is not vendor bound. The underlying service and Terraform manifests are specific to Azure, the application manifests and anything deployed to the cluster are agnostic. If needed, the client can change providers or even bring it back to on premises if they prefer. Provided there is a functioning Kubernetes service available, the codebase can be redeployed with greater ease compared to attempting to lift and shift your services if they were deployed on VMs. This concept was highlighted by Kerner (2018) ”While Kubernetes was born at Google, it is not tethered to Google and it can run anywhere, be it an on-premises server or a cloud data center.”

The ArgoCD dashboard provides an excellent visual representation of applications running on Kubernetes. Additional clusters can be added, and the UI can be broken down into projects, clusters or namespaces.

Future imrpovements

Kubernetes is a huge ecosystem and there are endless possibilities if this approach is embraced. The client, if they decided to invest in this idea and had the expertise to do so, could manoeuvre into a position of continuous integration and deployment. There is the potential to move away from Terraform as the infrastructure deployment tool and onto Crossplane, for example. Crossplane when combined with ArgoCD is incredibly powerful as the infrastructure is maintained in a real time basis. Another upside to moving in this direction is manifests are in YAML format and moves away from having to rely on HCL (or JSON). Instead of updating a complete cache like Terraform does, Crossplane can update and maintain individual components at any time. This is much more efficient than maintaining a collection of smaller Terraform manifests and running them on a schedule, as stated by Beetz & Harrer (2021)”The pull-based approach has several benefits over the push-based version. For one, the continuously pulling operator can observe the

state of infrastructure, react to deviations described by the environment repository, and deploy the infrastructure again." Here is a short comparison between Crossplane and Terraform.

```
    resource "azurerm_kubernetes_cluster" "default" {
  name                = "${random_pet.prefix.id}-aks"
  location            = azurerm_resource_group.default.location
  resource_group_name = azurerm_resource_group.default.name
  dns_prefix          = "${random_pet.prefix.id}-k8s"

  default_node_pool {
    name            = "default"
    node_count      = 2
    vm_size         = "Standard_B2s"
    os_disk_size_gb = 30
  }

  service_principal {
    client_id     = var.appId
    client_secret = var.password
  }

  role_based_access_control_enabled = true

  tags = {
    environment = "Demo"
  }
}
```

And a similar deployment for crossplane would be;

```
apiVersion: compute.azure.crossplane.io/v1alpha3
kind: AKSCluster
metadata:
  name: prod-cluster
spec:
  location: northeurope
  version: "1.25.0"
  nodeVMSize: Standard_D2_v2
  resourceGroupNameRef:
    name: k8s_testing
  dnsNamePrefix: k8s
  nodeCount: 2
```

YAML is much more readable than HCL and it is also vendor agnostic, with HCL being a Hashicorp specific configuration language. It would be expected that YAML is much more widely understood.

I may be extolling Crossplane but there is one huge caveat, and this is the main reason why I did not go down this route as the proposed solution. It is developing quickly, syntax and related CRDs (Custom Resource Definition) are liable to

change rapidly. Terraform is much more mature and in turn has a much larger resource library to avail of which provides greater stability in a production environment.

Other solutions

Before I arrived at the proposed solution I had considered a few other methods. One of which was to use Terraform to deploy VMs running Docker Swarm onto a cloud provider and attempt to utilise a blue/green deployment. This would mean new deployments (green) could fall back to the old deployment (blue) quickly in the event of problems. The main issue I had with this idea was the fact that the Terraform manifests would be much larger and much more complex. Ansible (or similar) would have to be used to deploy the Docker software and the Docker Swarm infrastructure would need to be planned and maintained. Autoscaling groups could be used in this scenario but with that comes increasing complexity and would do nothing to reduce the amount of time spent maintaining the infrastructure. This method would also require a large amount of resources as there still needs to be test and staging environments and then a replica of the production environment to enable blue/green switching.

Conclusion

The proposed solution attempts to empower the developers to be able to deploy and scale as they need without engaging other teams or spending large amounts of time maintaining their infrastructure. This comes at a cost, however, in the form of the steep learning curve with Kubernetes. Having a cloud managed service removes a lot of the infrastructure headache involved in such a setup, but as it scales and grows in complexity then deeper knowledge of the Kubernetes ecosystem will be required.

While Terraform is a declarative tool, it requires a means of triggering the workload, usually in a CI/CD pipeline as shown in my solution. Problems can also arise with the state file, it has to be stored somewhere, it is locked when Terraform is performing any kind of work which in turn prevents others from making changes to any related infrastructure. As deployments get larger they also take much longer and while Terraform workloads can be broken down into smaller projects it can become cumbersome depending on how the repository is maintained. There is also the chance of users deploying Terraform manifests from their local machine which would inevitably cause drift between repository and live environment. When implemented in a meaningful manner, Terraform is extremely capable and the Terraform resource library covers a wide range of service providers.

There is a payoff as this method addresses the clients concerns by providing a secure, self-contained environment. Infrastructure is stored as code, in a centralised repository and there is little risk of divergence or custom configurations on VMs or servers causing extensive maintenance time. The applications are constantly maintained by employing Continuous Delivery with ArgoCD. This prevents drift as the configurations are declarative and constantly maintained.

References

Alonso, J., Piliszek, R. & Cankar, M. (2022), 'Embracing iac through the devsecops philosophy: Concepts, challenges, and a reference framework', *IEEE Software* **40**(1), 56–62.

Beetz, F. & Harrer, S. (2021), 'Gitops: The evolution of devops?', *IEEE Software* **39**(4), 70–75.

Kerner, S. M. (2018), 'Top five reasons why kubernetes is changing the cloud landscape.', *eWeek* p. N.PAG.
URL: *https://tinyurl.com/mwcd2tc7*

Appendix

Link to a zipped copy of the GitLab repository and video presentation
https://tudublin-my.sharepoint.com/:f:/g/personal/x00205790_mytudublin_ie/Eg4WU43bttFPtP-oX2vt6LIBLtBMzckdppncQqoTI-J86w?e=lkxCLD

Teraform documentation on azurerm resources
https://registry.terraform.io/providers/hashicorp/azurerm/latest/docs/data-sources/resources

A blog post about Crossplane
https://blog.crossplane.io/why-crossplane-is-so-exciting/