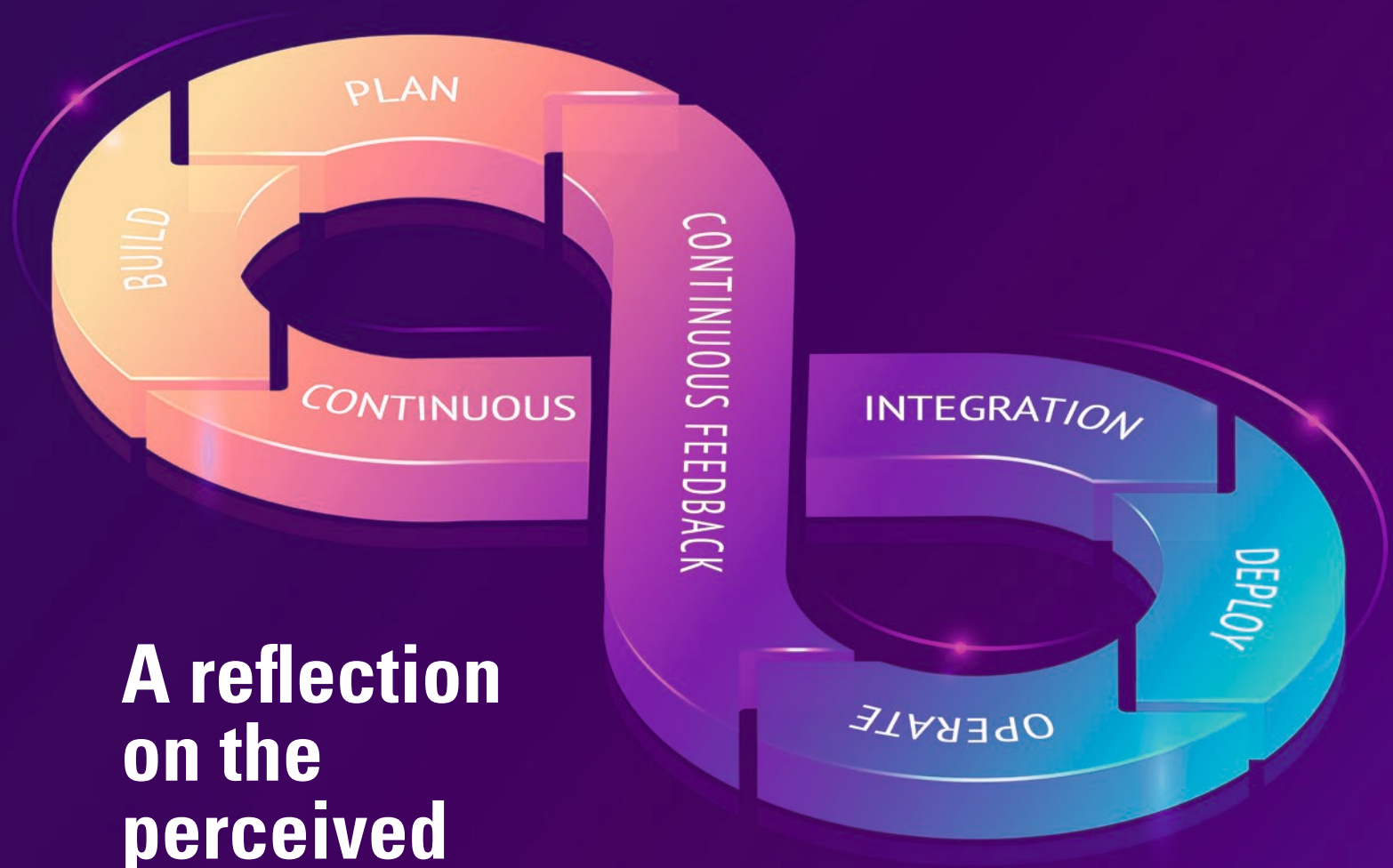


A reflection on the perceived benefits of Infrastructure as Code

Ir. René Pingen
is a senior manager at KPMG Advisory.
pingen.rene@kpmg.nl

A concrete case study to reflect on the value and costs of Infrastructure as Code

In this article we reflect on the value of applying Infrastructure as Code (IaC) for the development of a cloud native application. IaC is often presented as an obvious methodology to pursue in the development of modern cloud native applications, in practice it can be complex and costly to implement and maintain. We present design principles and good practices to consider in Infrastructure as Code implementation, and share lessons learned through the implementation of a concrete use case. Finally, we present some key considerations to determine if Infrastructure as Code is suitable for your project and/or organization.

Many attempts at IaC do not seem to achieve the perceived benefits: they are not fully automated, complex, difficult to maintain and/or understand

INTRODUCTION

The concept of Infrastructure as Code (IaC) is often perceived as the holy grail of cloud computing and DevOps. The idea of automating everything to eliminate manual activities in the deployment and maintenance of application infrastructure and reducing operational costs, is definitively something worth pursuing. In practice however, we see many attempts at IaC that do not seem to achieve the perceived benefits: they are not fully automated, complex, difficult to maintain and/or difficult to understand (and thereby leading to higher costs). In our view, the concept of IaC goes beyond automation. It is about treating infrastructure as software, meaning that the development and maintenance of IaC should follow software engineering methodologies and best practices.

Infrastructure as Code is the practice of applying software engineering methodologies on the development of infrastructure. Infrastructure configurations are defined as code, which is managed using software versioning tools (e.g. GIT) and automatically deployed using DevOps and CI/CD processes and tools.

In this article we will describe our experiences in applying IaC in the context of a multi-tier cloud native application developed on the Microsoft Azure platform, and reflect on the value of using IaC versus the costs (initial implementation and change costs over time). We will start with some context regarding the deployment of IaC models, followed by introducing our case study. Next, we will provide the key design principles and design decisions, discuss the key challenges we faced in the implementation and finally reflect on the value versus the costs.

SELECTING THE RIGHT TECHNOLOGY AND DEPLOYMENT MODELS FOR INFRASTRUCTURE AS CODE

Organizations and DevOps teams are applying IaC in different ways. There are different technologies that can be used to deploy Azure resources, including:

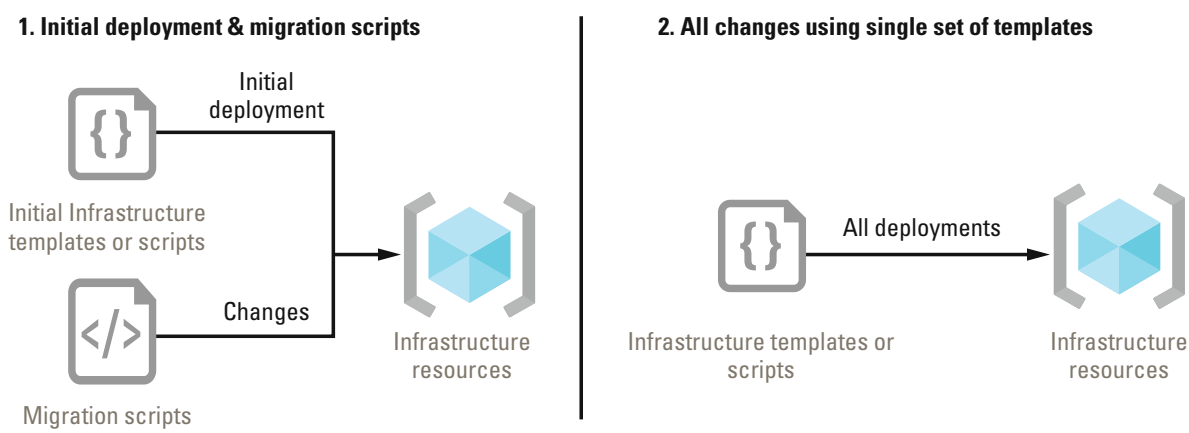
- Template based languages such as ARM templates and TerraForm (declarative)
- Azure CLI and PowerShell scripts (Imperative)

Note that for different cloud platforms (AWS or GCP) similar technologies are available.

Even though this has been debated, both the declarative template languages (ARM and TerraForm) and the imperative scripts (Azure CLI and PowerShell) are in essence idempotent. This means that redeploying the templates/scripts multiple times will lead to the same result. In practice we see that these templates or scripts are used in two different ways:

1. *Initial deployment templates and migration scripts.* IaC templates or scripts are used for the initial creation

Figure 1. Infrastructure as Code deployment models.



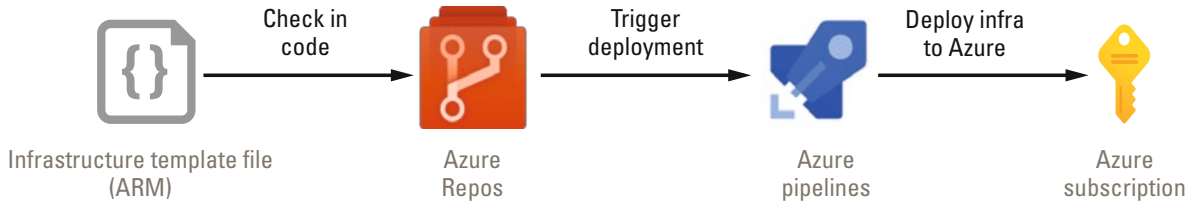


Figure 2. Infrastructure as Code deployment pipeline.

(day 0) of resources (e.g. VM's, databases). After this point, changes to the components are made using migration scripts. Note that in most cases this does mean that the original template cannot be used after the initial deployment, because any changes made after the initial template are not reflected in the template.

2. *All changes using a single set of templates.* Using an idempotent deployment template that contains the full configuration of the resources in scope. All configuration changes need to be made in the template, which is then redeployed. The advantage of this method is that the entire configuration is in a single place, which simplifies future deployments and potential migrations. Note that in this case, any configuration setting that is not specified in the template will be overwritten by the deployment.

We applied the second approach in our case study. This way, we ensure that we always have an accurate representation of the infrastructure configuration and that there is consistency between the different components that need to connect. This requires that we are confident that no other changes are made to the environment and that the template is fully idempotent to avoid downtime. In practice, we see that many DevOps teams do not trust the templates and deployment mechanism to apply this methodology (see the following section on deployment anxiety).

CASE STUDY: DEPLOYING A CLOUD NATIVE APPLICATION THROUGH IAC

Our application is an advanced analytics application that consists of several components: a web-based client, database, data lake that contains several datasets, scalable compute capacity to run algorithms and ETL tooling to move and transform data.

The infrastructure of our application has been developed on the Azure cloud platform by following a set of IaC principles and design decisions. We use Azure Resource Manager (ARM) templates for the definition of the infrastructure, and Azure DevOps as CICD platform to deploy the infrastructure to Azure.

The high-level flow is illustrated in Figure 2. We develop the IaC templates, check them into an Azure GIT repository, which then triggers an Azure deployment pipeline. This deployment pipeline deploys the Azure resources to the specified environment according to the specified templates.

With this approach, we automate the process of deploying the infrastructure, and use automated test cases to perform sanity checks and ensure that the infrastructure is operational.

PRINCIPLES AND DESIGN DECISIONS

Inspired by concepts such as the twelve factor app ([Wigg17]), SOLID principles for object-oriented design ([Mar03]) and others ([Bion19], [Feda19]), we have selected a number of principles and design decisions that we typically follow in the implementation of our solutions and have followed for this case study.

Infrastructure as Code templates and deployments are idempotent

A function is called idempotent if applying the function multiple times leads to the same result. In the context of IaC, this means that deploying infrastructure templates will always lead to the same result. If we for example want to change a configuration setting of an already deployed database resource, we can simply change the configuration setting in the initial template, redeploy the entire template which results into one database with the right configuration. In our case, we did have to deal with a couple of limitations to ensure the infrastructure templates remain idempotent (for example: deployments of code can lead to infrastructure configuration changes that need to be handled in the infrastructure deployment).

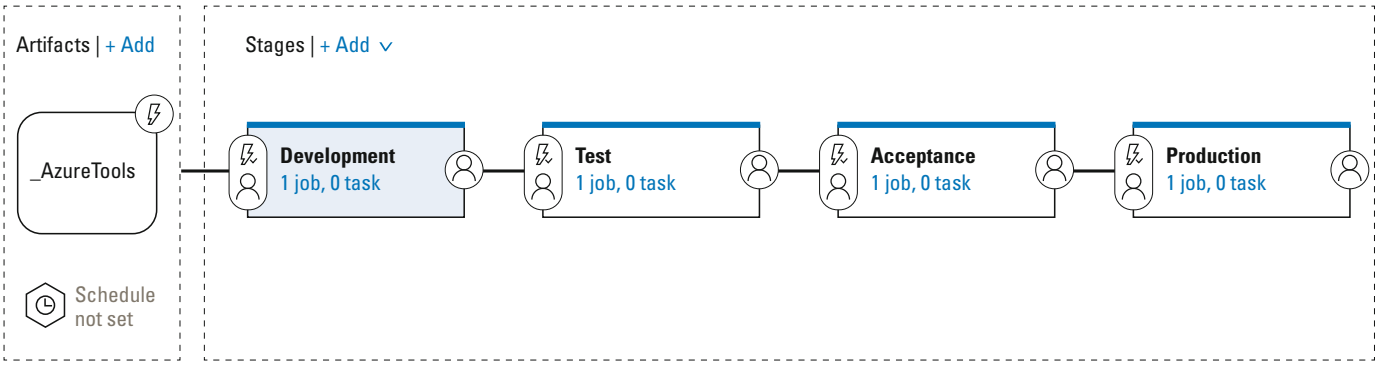


Figure 3. Deployment pipeline in Azure DevOps.

The infrastructure is fully managed through code and under source control

In traditional environments, the concept of configuration drift is a serious concern. Configuration drift is the concept where the actual configuration of production applications and servers changes over time while the design and documentation are not updated, meaning there is no complete documented to-be configuration. This can lead to tricky situations, for example in migration scenarios where no one remembers how the application should be configured. By not allowing write access to the environments (typically acceptance and production), we can enforce that all changes are made through code. All code is under source control (in our case GIT) to enable versioning of the infrastructure templates, which ensures that the full configuration is captured in code (although supporting documentation is still highly recommended).

Secrets are stored in vaults, configuration is separated from code

The ARM templates to deploy the infrastructure depend on configuration settings and secrets. Secrets should be stored in KeyVaults and can be passed to templates using KeyVault references, as documented by Microsoft ([Micr20]). Environment configuration settings should be passed to the templates at deployment time.

Keep it simple

In essence, the infrastructure templates and deployment pipelines should be simple to use and maintain. In practice, this can be difficult as environments can grow complex and in some cases workarounds need to be applied (see challenges section). To keep the environment simple, we have followed a couple of guidelines:

- *Only pass environment specific, user friendly parameters to the templates.* Limit the number of parameters that should be passed to the templates to keep usage simple. For example: we pass the parameter 'env' to the templates which we use to generate the names for all the resources, instead of passing all resource names individually.
- *Keep the templates modular and maintainable.* Putting all resource definitions in one big template makes them complex to maintain in the long run. Through the use of linked templates (see picture below), it is possible to modularize the infrastructure definition.

Dev/prod parity

In order to properly test deployments before going to production, it is important to ensure that development, test and acceptance environments are (nearly) equal to production. This means that the same templates should be used to configure the environments and differences between the environments should be kept to a minimum (e.g. production environment is differently sized). Through the use of CI/CD deployment pipelines, we can ensure that infrastructure deployments are deployed first to the test and acceptance environments before being deployed to production, see the example Azure DevOps pipeline below

The solution infrastructure is deployed with a linked scripts/templates in a single pipeline

We have seen many implementations in Azure where separate ARM templates are deployed in individual pipelines. Dependencies between the different components are manually passed as configuration parameters, leading to a complex set of templates and pipelines, which is difficult to maintain. By using one integrated master template for the solution, we manage all these dependencies centrally, ensuring references to other components are valid and naming is consistent.

Note: In some cases, this is not feasible in practice, or workarounds are required, but the principle holds in general.

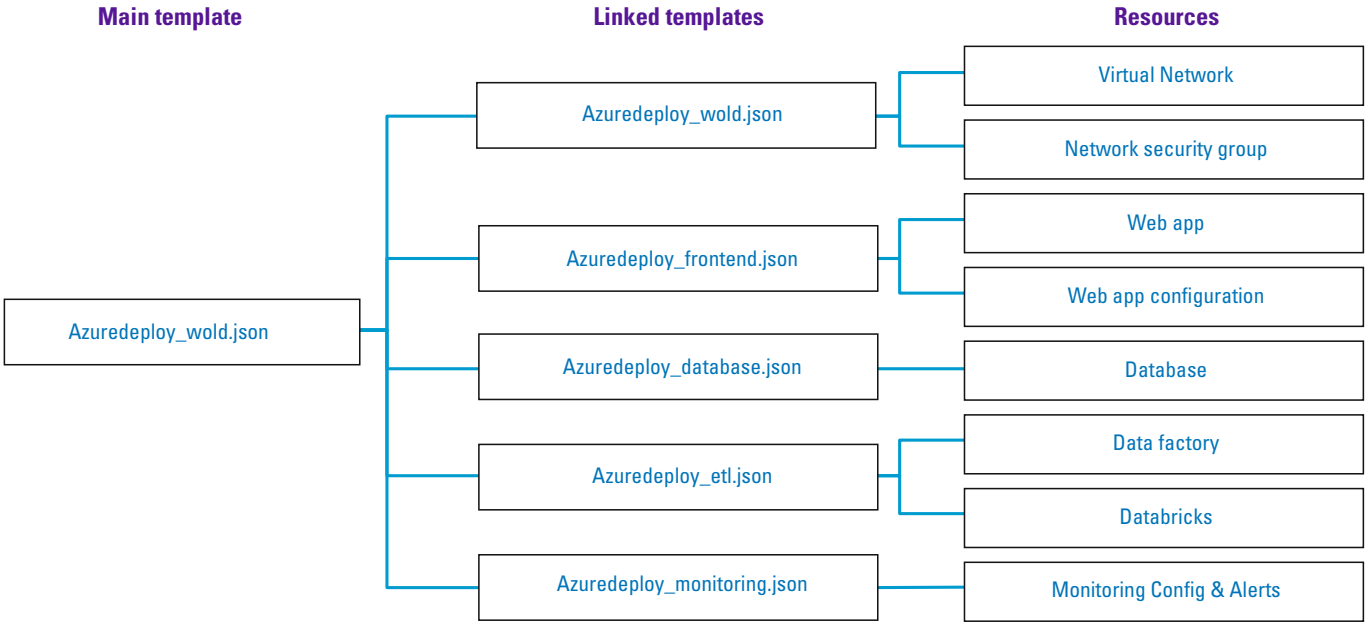


Figure 4. Using nested Azure Resource Manager (ARM) templates to deploy multiple resources.

BRINGING EVERYTHING TOGETHER IN OUR CASE STUDY

For our multi-tier cloud native application, we managed to create a full declarative configuration of the infrastructure. All resources can be deployed from a single template through the use of linked templates, as depicted in the image below.

The deployment of the infrastructure is performed in two steps:

1. The deployment of the 'supporting infrastructure' which includes:
   - KeyVault that will be used to store all secrets (e.g. generated database passwords)
   - Storage account that is used to store the linked templates ([Micr21])
2. The deployment of the entire solution infrastructure (deploy_world).

These two deployment steps are performed for each environment (dev, test, acceptance and prod) with the applicable parameters.

CHALLENGES AND LESSONS LEARNED

In the development of the solution, we faced a number of challenges and issues, which led to a couple of insights and additional design decisions:

1. Developing clean deployment templates

As cloud technology is rapidly evolving and new features are quickly released, documentation is unfortunately not always up to date or complete. In the development of our templates, we sometimes had to compensate for missing documentation by searching for templates published online, using the 'export template' for ARM templates in the Azure portal or even sniffing API calls to Azure to find out the right way to write the templates. Especially these last two options can be very tricky and - in some cases - lead to unreliable templates. Our key takeaways:

- Use an exported template for inspiration when needed but always write the template yourself. This ensures the template is free of 'clutter' and you understand what is in the template.
- IaC is (a form of) software engineering. Use an integrated developer environment (IDE) such as Visual Studio Code to support and speed up development.

2. Selecting the right modularity of deployments

Just as in designing other software systems, finding the right way to modularize infrastructure templates is an art in itself. In essence, it is about modularizing where it makes sense, and keeping things together that belong together. Taking into account software design principles such as SOLID ([Mart03]) and concepts such as domain-driven design, we follow the following principles:

- a resource group contains resources with the same lifecycle

- an infrastructure template/script contains resources that logically belong together
- split up large templates in logical building blocks where possible
- limit parameter passing between templates and scripts
- templates can be deployed individually
- map dependencies explicitly, and do not allow circular dependencies between templates

3. Avoid deployment anxiety by deploying frequently

As described in the first section of this blog, we have opted to use a full deployment of the infrastructure using a single declarative template. There are several reasons why this mechanism can break (over time), for example:

- *Application deployments that influence infrastructure settings. I*n our deployment we found that the deployment of application code in some cases changes some application configuration settings that are overwritten when the infrastructure is redeployed. This can leave the environment in an inconsistent state.
- *Manual actions in the infrastructure that influence infrastructure settings.* In some cases, manual actions are needed in the infrastructure that for some reason cannot be automated. In our case we had one manual action related to DataBricks that cannot be automated (yet). To avoid undoing this manual action (and thereby breaking the environment) with a redeployment of the infrastructure, we had to introduce a workaround in the templates to ensure the environment remains stable after redeploying the infrastructure.
- *Access permissions or access keys change over time. In* larger organization access keys and permissions are often provided by another team. These dependencies on other teams can lead to faulty deployments over time (e.g. key expirations or access right changes). It is important to acknowledge these dependencies and handle them in your code.
- *Other unexpected behavior (e.g. changes in the services provided by the cloud platform).* This is the concept which DevOps engineers fear the most, but in our experience this is rarely the actual cause for issues.

The potential causes for issues above lead to the concept of deployment anxiety: losing the confidence of deploying to (production) environments. Based on our own experience, this fear is a serious concern when working with infrastructure as Code, even when extensive automated testing procedures are in place.

Infrastructure configuration tends to stabilize in the project, even when the application is under active development. This also means that some of the more fundamental principles of agile and DevOps with regard to Continuous Integration and Continuous Delivery do not fully apply. Those concepts work when you deploy small changes fast and frequently, supported through automated testing. If no changes are made to the infrastructure for 6 months and therefore no deployments have been performed, deployment anxiety tends to increase. Deploying frequently reduces deployment anxiety, which can be done in two ways:

1. Scheduled (Daily/Weekly) deployments of the latest (changed or unchanged) version of the code to acceptance and production to prove the deployment is still valid and effective.
2. Combining infrastructure and application deployments in a single pipeline to ensure infrastructure is deployed more frequently.

In both cases, a mature deployment pipeline that includes automated testing is required to ensure the deployments work and the environment remains stable.

CONCLUSION: IS IAC WORTH THE INVESTMENT?

As the challenges described above illustrate, reliable IaC implementations are tricky. They require a good design based on a clear set of principles and a consistent implementation. The question we always try to answer: is the added value of IaC implementations worth the investment? Many DevOps advocates will undoubtedly say yes, but the honest answer is: it depends (we are engineers after all).

The first question is which technology provides the best value. ARM templates can become quite complex to build and maintain, and other technologies such as Azure CLI or TerraForm might provide advantages with regard to readability and maintainability ([Nabe18]). The technology choice is an important one to make based on the experience, capabilities and demands of your organization.

Reflecting on the two deployment models in Figure 1, we can evaluate the added value of IaC. In the first model, templates can only be used in an initial deployment. This means that the return on investment is only positive when the time saved on initial deployments outweighs the IaC implementation cost. In other words: how often do you need to deploy new instances of the environment or how frequently do you start with a fresh environment (e.g. migration)? An example can be a cloud competence center that defines standard VM

templates for all DevOps teams to use. For an application with specific templates which is expected to be only deployed 2-3 times in its lifetime, designing such an initial deployment template is probably not worth the investment.

The second model has a slightly different use case. The required investment to do this properly is higher, but doing it right delivers more value than the first model. The return on investment is not only determined by the number of times the environment is redeployed, but also all the maintenance effort it (potentially) reduces.

Bringing this together there are three main components to consider:

1. *Effort required to build the templates, scripts and pipelines.* The effort to properly design re-usable IaC templates, scripts and pipelines can be quite significant, and is often underestimated. Experience and good design decisions will reduce these efforts over time.
2. *Manual maintenance effort reduced through re-use.* If a template for a virtual machine is used throughout a company, using IaC will greatly reduce the time required to deploy new infrastructure. If a template is specific for one application this is not the case.
3. *Reduced maintenance efforts and increased stability and security as a result of standardization.* When properly implemented, IaC enforces a level of standardization that can lead to reduced maintenance efforts (e.g. updating many components at once through a single template or pipeline) as well as increased stability and security. This does however require a mature organization and commitment to work consistently according to a set of principles.

In our work, we try to determine the best approach on a case-by-case basis, although we do see that experience with the second model will reduce the effort for future implementations. During this implementation, we resolved several complex issues that we will be able to take with us in future solutions, reducing the time needed for implementation. On the other hand, handing over this knowledge to new engineers or teams can be difficult.

Regardless of the model you select, make sure you select the model consciously, follow a clear set of principles and implement it consistently.

References

[Bion19] Biondic, D. (2019, 25 January). 7 Principles of Infrastructure as Code (on Azure and beyond). Retrieved from: https://blog.coffeeapplied.com/7-principles-of-infrastructure-as-code-on-azure-and-beyond-51842e13b00

[Feda19] Fedak, V. (2019, 7 October). Infrastructure as Code DevOps principle: meaning, benefits, use cases. Retrieved from: https://medium.com/@FedakV/infrastructure-as-code-devops-principle-meaning-benefits-use-cases-a4461a1fef2

[Mart03] Martin, R.C. (2003). The Principles of OOD. Retrieved from: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

[Micr20] Microsoft (2020, 17 December). Use Azure Key Vault to pass secure parameter value during deployment. Retrieved from: https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/key-vault-parameter?tabs=azure-cli

[Micr21] Microsoft (2021, 12 February). Tutorial: Deploy a linked template. Retrieved from: https://docs.microsoft.com/en-us/azure/azure-resource-manager/templates/deployment-tutorial-linked-template

[Nabe18] Naber, P. (2018, 11 November). Stop using ARM templates! Use the Azure CLI instead. Retrieved from: https://pascalnaber.wordpress.com/2018/11/11/stop-using-arm-templates-use-the-azure-cli-instead/

[Wigg17] Wiggins, A. (2017). The Twelve-Factor App. Retrieved from: https://12factor.net/

About the author

Ir. René Pingen is a senior manager and cloud architect at KPMG. He has a background in computer science and performed numerous engagements in the fields of IT strategy, architecture, software, cloud and security. René has been named Cloud Architect of 2020 by the Cloud Architect Alliance.