SILESIAN UNIVERSITY OF TECHNOLOGY PUBLISHING HOUSE

SCIENTIFIC PAPERS OF SILESIAN UNIVERSITY OF TECHNOLOGY 2023
ORGANIZATION AND MANAGEMENT SERIES NO. 186

CURRENT INFRASTRUCTURE AS A CODE AUTOMATION TRENDS IN CONTEXT OF CLOUD AGNOSTIC RESOURCE PROVISIONING

Ireneusz J. JÓŹWIAK[1], Piotr P. JÓŹWIAK[2*], Krzysztof ZATWARNICKI[3]

[1] General T. Kościuszko Military University of Land Forces in Wrocław; ireneusz.jozwiak@awl.edu.pl, ORCID: 0000-0002-2160-7077
[2] Wroclaw University of Science and Technology; piotr.jozwiak@pwr.edu.pl, ORCID: 0000-0002-5325-3728
[3] Opole University of Technology; k.zatwarnicki@po.edu.pl, ORCID: 0000-0001-6156-6030
* Correspondence author

Purpose: The aim of the research is to determine the maturity of the available tools for building Software as a Service (SaaS) services that enable automation of deployment to multiple cloud operators using a single infrastructure definition known as Cloud Agnostic.
Design/methodology/approach: The paper related to the development of areas of software engineering has been the automation of processes for building, testing, integrating, and delivering applications developed by large development teams. It has come to be known as continuous delivery process. We provided an overview of the tools available to automate infrastructure provisioning in Cloud Agnostic manner.
Findings: The research indicated that there are solutions on the market for building automation of cloud infrastructures, however, most of these are not geared towards achieving the Cloud Agnostic definition. One tool called Crossplane was researched, which was designed from the outset to enable Cloud Agnostic definitions for infrastructure provisioning. The research has shown that, as of today, the Kubernetes platform with an extension of Crossplane is the best approach to enable a loose attachment to a single cloud operator.
Originality/value: The proposal to use the Kubernetes platform with additional tools significantly reduces the risk of strong attachment to single operator cloud solutions. The proposed design approach can be helpful for IT system architects in decision making.

Keywords: strategy of infrastructure provisioning, infrastructure automation, software engineering, tool, Cloud Agnostic resource definitions.

Category of the paper: Research paper.

1. Introduction

In recent years, a strongly developing area of software engineering has been the automation of processes for building, testing, integrating, and delivering applications developed by large development teams. Work related to the development of these areas has come to be known as

continuous integration/continuous delivery processes, abbreviated CI/CD. Nowadays, it is no longer sufficient to compile source code into a form suitable only for traditional distribution, e.g. in the form of executable binary files, because the distribution process has changed significantly. In many cases moving towards a *Software as a Service* (SaaS) sales model. The ability to build SaaS applications is closely linked to the ever-increasing popularity of public cloud services provided by major IT players such as Google, Microsoft, Amazon, and Oracle. Many companies tie their commercial success directly to deploying their services on these environments rather than building their own on-premise computing center. This type of approach unlocks significant potential for companies that do not need to have large financial resources at the outset. Cloud services make it possible to spread costs over time and match them closely to current demand, increasing the scalability of investments. The use of public data centers eliminates the need to purchase hardware with a stockpile to ensure uninterrupted continuity of operation with the increasing volume of traffic generated by the customer of a given service.

2. Models of Infrastructure as a Code

The increased interest in cloud computing and the automation of software delivery has forced the cloud market to make available a suitable API and SDK to define hardware resources as code. Tools related to the automation of hardware resource orchestration have been called *Infrastructure as a Code* (IaaC). With usage of many programming languages, IaaC is responsible for provisioning and managing resources in data centers. The main premise of IaaC is to completely eliminate the manual provisioning and configuration of all resources by humans. This is intended to minimize human error and thus minimize the risk of errors in the application environment. Defining infrastructure as a code also ensures that complex enterprise execution environments can be built in a consistent, automated, fast and testable manner. IaaC automation unlocks human resources that can be allocated to other business tasks.

There are two approaches for building IaaC, closely related to the available programming paradigms (ScriptRock, 2015):

- *declarative/functional* approach,
- *imperative/procedural* approach.

The declarative approach involves providing the configuration in the form of a description of *what we want to have*. It is the task of the process performing the automation in question to know how to do it. The opposite approach is the imperative approach, which focuses on describing *how to get to the desired state instruction by instruction* (Loschwitz, 2014). So, in this approach, the programmer uses specific procedures that transform the environment to its final state.

Both approaches for building IaaC have their advantages and disadvantages. The imperative approach offers greater control over the automation process. The imperative language expression provides the necessary programming structures to allow alternative execution of specific procedures. However, in this approach it is very difficult to determine what the target infrastructure configuration should look like. The only way to determine this is by tracing the source code and trying to understand how it works. This problem does not occur with the declarative approach. As the name suggests, this approach inherently defines what we want our environment to look like, without providing instructions on how to get there. This description is completely devoid of instructions defining how to get to that state. The IaaC runtime environment hides the details of the execution of the definition. On the one hand, this is a very tempting assumption that naively relieves us of the compulsion to know how the process works. However, in practice it is often the case that the tool stack in a state where it is unable to transition to a desired new state. In such a situation, the team is relied upon to find a suitable solution to the problem, which many times provides to manually streamlining the process ad-hoc. This is an undesired situation, but unfortunately one that realistically occurs in practice.

Very important issue of IaaC is the problem of defining infrastructure in *Cloud Agnostic* manner. The Cloud Agnostic process has the basic characteristic that one common definition is independent of the specifics of a particular cloud service provider. This is a state that is very difficult to achieve, often due to differences in philosophy and tools provided by individual providers. The primary task of building Cloud Agnostic tools is to prepare IaaC definitions that loosely link us to one provider, allowing us to quickly convert to another provider. In an ideal approach, often not fully achievable, Cloud Agnostic assumes that a single code will work for all platforms (Copado, 2022).

3. Synergy of DevOps teams

The emergence of IaaC processes has also had a significant impact on the organization of the software development and maintenance teams themselves. The work of IT administrator's teams, hitherto understood as imperative, manual control of infrastructure configuration through partial automation in scripts, is slowly being transformed into a nature closer to that of IT developer's teams. This is becoming possible because IaaC and CI/CD, at its foundation, insists on replacing these practices in favor of a full, consistent description in the form of code, which will not be executed directly by humans, but by automations such as, for ex. Jenkins pipelines (Kim, Humble, Debois, Willis, 2016). This has led to the emergence of a new software development methodology called *DevOps* (Azad, Hyrynsalmi, 2023). This methodology recognizes the product as something broader than just software development, also including the processes of software integration, deployment (alternative delivery), maintenance in the

definition of the product. This methodology strongly unifies two teams hitherto seen as separate, creating multidisciplinary teams holistically responsible for the entire process of software development, testing and running in target environments (Kim, Humble, Debois, Willis, 2016).

Automation is a key enabler of business success, according to a study by Dynatrace, published in 2022 by CISO REPORT (Ciso Report, 2023). As many as 90% of the organizations surveyed indicated that the pressure for digital transformation has increased significantly in the last 12 months. At the same time, only 34% of the organizations surveyed have mature DevOps teams, while as many as 55% of organizations face tradeoffs among quality, security, and user experience to meet the need for rapid transformation (Ciso Report, 2023). These studies clearly indicate that the trend of building business success is strongly linked to the introduction of the DevOps model into an enterprise organization.

4. Heterogeneity of Infrastructure as a Code

The emergence of the concept of describing infrastructure in the form of code executed by computers has opened the path of rapid deployment, reducing the time from the publication of new functionality in the code repository to deployment in production to as little as several minutes. Once the changes have been committed to the code repository, the relevant processes, known as pipeline, run automatically testing the quality of the code, provisioning a temporary test infrastructure, and at the end instantly deliver the software to production environment.

The main problem with IaaC is finding the tools to create the required resources, often in heterogeneous environments, which can be a challenge. Typically, meeting a rapid implementation of DevOps methodologies that is cost optimal involves moving the on-premise infrastructure to the cloud. This is not an easy decision, with many factors to be analyzed that affect the ultimate success, from the obvious in the form of cost, to the availability of the necessary resources from a given cloud provider. Also important are legal regulations, forcing, for example, the storage of data in specific regions of the world, and the expectations of end customers, who often only agree to sign a contract if the SaaS hosting infrastructure will be in a specific cloud.

The foundation needed to describe the infrastructure in form of code is for the cloud provider to provide the appropriate tools. In the next chapter, we focus on briefly characterizing the available solutions used in the implementation of IaaC.

5. Cloud native tools for infrastructure automation

Without the right tools provided by the cloud provider, it is impossible to think seriously about infrastructure automation. They act as a fundamental doorway into the cloud, enabling developer interaction with resources. In this paper, we provide a brief overview of these tools available in the three most popular clouds: *Amazon AWS*, *Microsoft Azure* and *Google Cloud*.

5.1. Cloud Command Line Interfaces

The primary tool for interacting with clouds is the *Command Line Interface* (CLI), accessible from the operating system command line. Its most common use is in various shell/bash scripts. From the point of view of infrastructure administrators, it is the most natural choice, as it fits directly into the tools that these teams use on a daily basis. The CLI allows quick interaction from the operating system command line but is also well suited to automating selected processes in, for example, Jenkins pipelines.

Amazon AWS makes the *AWS CLI* (AWS CLI, 2023) available to users in two versions. The newer v2 version is a more extensive offering of its predecessor. The tool is available for all popular operating systems, like *Windows*, *macOS* and *Linux*. It is also available as a *Docker image*, removing the need to install the tool directly on the system. The AWS CLI delivers high functional coverage, allowing configuration and management of almost all offered AWS services. The AWS CLI also allows control over the output format, greatly enhancing the tool's ability to be used in scripts. Both human-readable and software-parsable formats are available: *JSON*, *YAML*, *YAML-stream*, *text*, *table*.

A similar tool is provided by Microsoft Azure, in the form of a CLI called ***az*** (Azure CLI overview, 2023). The tool is available for all leading operating systems like *Windows*, *macOS*, *Linux* and as a *Docker image*. It is also possible to use directly from a web browser in a service called *Cloud Shell*. Coverage of functionality is very high. However, a lot of functionality requires the installation of appropriate extensions called features. This can be inconvenient when writing automation scripts, as you always have to remember to install all the features you will need in the script. Unlike AWS CLI, Azure az has self-upgrade functionality.

The last featured service provider Google Cloud also has a CLI called *gcloud* (Install the glocud CLI, 2023). The tool also has very high functional coverage. It is available like its predecessors for all the platforms mentioned, including as a *Docker image*.

To some extent, each CLI reflects the ethos of their cloud. The AWS CLI is dense, powerful, and occasionally inconsistent. The Azure CLI is rich, easy to get started with, and sometimes more complicated than it should be. And the Google Cloud CLI is clean, integrated, and evolving. However, the differences in these tools and the shell character makes them ultimately a poor fit for mature Infrastructure automation solutions using IaaC. Shell scripts are difficult to analyze and document. Any corrections can be erroneous. Of course, there is no way

to have a Cloud Agnostic solution where one script can execute on all clouds. Wanting to cover multiple clouds we are forced to write multiple versions of scripts.

5.2. Cloud Software Development Kits

While in the case of the CLI, all platforms provide very similar functionalities, the case is more diverse in the case of the SDK. What is an SDK? An SDK is a *Software Development Library* prepared for a specific language or framework. An SDK allows the infrastructure to be defined in the form of imperative code.

Amazon AWS provides SDKs in as many as twelve programming languages, such as (Developer Tools, 2023): *Python*, *JavaScript*, *PHP*, *Java*, *C++*, *NodeJS*, *Go*, *Ruby*, *.Net*, *Kotlin*, *Rust*, *Swift*. However, AWS also provides specific SDKs for *web development*, *mobile development*, or *IoT*.

Microsoft Azure provides SDKs in languages such as Azure-sdk repository (Azure-sdk, 2023): *.Net*, *Go*, *C*, *C++*, *Java*, *JavaScript*, *Python*. Additionally, as with AWS, it provides specific SDKs for *Android* and *iOS*.

Google Cloud, also provides an SDK, but it works a bit differently. What Google calls the Cloud SDK is for using the gcloud CLI tool, and if you want to use a specific language or platform with GCP, then you use one of the hundreds of Google APIs (Google Cloud SDK, 2023). At the same time, Google provides libraries to support interaction with the APIs in languages such as *Java*, *Go*, *Python*, *Ruby*, *PHP*, *C#*, *C++*, *NodeJS*.

Providing SDK libraries to support interaction with cloud computing greatly facilitates the automation of the infrastructure, even allowing the relevant code to be embedded along with the application code. The application itself is given the ability to be aware of where it is installed and the state of the infrastructure. However, all the solutions mentioned above do not allow Clous Agnostic IaaC to be written easily. The differences between the libraries are very large and, in the case of Google cloud, they already differ at the level of operating philosophy. Thus, a solution using the SDK directly forces multiple implementations for each cloud separately. Therefore, achieving Cloud Agnostic IaaC is very expensive.

As with the CLI, answering the question of what the infrastructure contains requires a tedious process of analyzing the source code. This is strongly related to the imperative/procedural nature of programmatic solutions.

5.3. Other cloud specific IaaC tools

The difficulty of analyzing imperative code has forced the development of solutions based on declarative code. Declarative notation is much easier for humans to understand and, above all, much more efficient. The definition is more concise and less error prone. The declarative solution ensures high reproducibility and modularity. The following shows which declarative tools are provided by the three cloud providers.

Amazon AWS provides *CloudFormation* (CF) template functionality in *YAML* or *JSON* format. This is the most supported tool for automating the orchestration of resources by AWS. A CloudFormation template is a declarative record of the list of resources and their configurations to be deployed in the cloud. CF provides an appropriate layer of parameterization and modularity to the templates so that they can be reused. An important advantage of CloudFormation is that it offers the deepest level of integration with the AWS cloud, including features like Designer, which lets you create and modify CloudFormation templates directly on the AWS website. However, there are times when small parts of the infrastructure configuration are not available in the CloudFormation template. An example of this is the inability to create an encrypted version of the SSM Parameter. Although the presented problem with the lack of 100% functionality coverage is found in all described tools. CloudFormation also provides a high level of assurance that your templates will always remain compatible with AWS services, even if Amazon makes changes to its services. An example of a CF template is shown in Table 1. This is an example template that creates a subnet.

Table 1.
Example of AWS and Google templates for subnetwork provisioning

# AWS Cloud Formation template AWSTemplateFormatVersion: "2010-09-09" Metadata: Generator: "notepad" Parameters: SubnetCidr: Type: String Default: "10.0.0.0/24" Resources: mySubnet: Type: AWS::EC2::Subnet Properties: VpcId: Ref: myVPC CidrBlock: !Ref SubnetCidr AvailabilityZone: "us-east-1a" Tags: - Key: stack Value: production	*# Google Cloud template* resources: - name: myNetwork type: compute.v1.network properties: autoCreateSubnetworks: true - name: mySubnet type: compute.v1.subnetwork properties: ipCidrRange: 10.130.0.0/20 network: $(ref.myNetwork.selfLink) region: us-central1

Source: own work.

In Azure, two solutions are available to the user. The first is the *Azure Resource Manager* (ARM) templates, enabling declarative description of infrastructure in *JSON* format. They are an equivalent solution to Amazon CloudFormation, providing similar functionality, including template parameterization. However, Microsoft has gone further and designed a second tool called *Bicep*, which is its own domain-specific language (DSL) solution that provides a declarative description of infrastructure. An important advantage of Bicep is its immediate support for new functionality emerging from Microsoft's cloud. As soon as new resource types and API versions are introduced by the vendor, they can be used in the Bicep file, without having to wait for the tools to be updated before working with the new services.

The language has a simple syntax and compared to a *JSON* template, is more concise and easier to read. An example of a Bicep script is shown in table 2. Presented script creates an example subnetwork. Due to space constraints for the article, an example of the ARM template is not included, as it is based on *JSON*, which by its nature is quite large in a human-readable format.

Table 2.
Example of Azure Bicep template for subnetwork provisioning

```
param location string = resourceGroup().location
resource virtualNetwork 'Microsoft.Network/virtualNetworks@2021-05-01' = {
 name: 'sarahs-network'
 location: location
 tags: {
 Purpose: 'Example subnet'
 }
 properties: {
 addressSpace: {
 addressPrefixes: [ '20.0.0.0/16' ]
 }
 subnets: [
 {
 name: 'mySubnet'
 properties: {
 addressPrefix: [ '20.0.0.0/24' ]
 }
 }
 ]
 }
}
```

Source: own work.

Google Cloud also provides a very similar mechanism to both predecessors in the form of scripts written in *YAML* format. This solution has been given the name *Deployment Manager* (DM) template. Table 1 shows a comparison of the Amazon CloudFormation and Google DM templates. Both scripts create a sample subnet.

The tools shown are very similar in many aspects. However, they are not tools that can be used between clouds, as they are vendor specific. Thus, they have the same problem as already discussed SDK tools. Achieving a cloud agnostic definition requires simultaneous description in all tools.

6. Multicloud IaaC tools

The tools presented in the previous chapter are solutions provided by cloud service developers, thus focusing only on interaction with a specific cloud. They are as sufficient as possible in a situation where an implementation is only planned for one specific cloud. In a situation where there is even a slight assumption that the application under development will be delivered to more than one cloud, or where we are not sure which cloud to choose,

the use of the tools described above will prove to be a significant limitation increasing the cost of the entire project. Today, there are tools that try to solve the above limitation. Tools such as *Ansible*, *Puppet* or *Terraform* have been on the market for many years.

Ansible is widely considered to be simpler. Puppet is model-driven and was built with systems administrators in mind. It follows a client-server (or agent-master) architecture. You install Puppet Server on one or more servers and then install Puppet Agent on all the nodes you want to manage. With both tools user can only provision a subset of available resources on particular cloud. Ansible and Puppet requires the installation of specialized agent software inside the cloud to operate/execute definitions.

Terraform is essentially the first tool to move significantly away from the pure context of administrative work and was designed with the broader DevOps context in mind. Terraform can manage infrastructure on all major cloud platforms. The human-readable *YAML* language helps write infrastructure code quickly. Terraforms state allows you to track resource changes throughout your deployments. For smooth operation, Terraform definitions should be written to the code repository along with the current state. This is related to Terraforms operating model, which saves locally executed operations and compares them with the current state in the infrastructure. If you're using Terraform for a personal project, storing state in a single terraform.tfstate file that lives locally on your computer works just fine. But if you want to use Terraform as a team on a real product, you run into several problems (Brikman, 2016):

- *Shared storage for state files.* To be able to use Terraform to update your infrastructure, each of your team members needs access to the same Terraform state files. That means you need to store those files in a shared location.
- *Locking state files.* As soon as data is shared, you run into a new problem: locking. Without locking, if two team members are running Terraform at the same time, you can run into race conditions as multiple Terraform processes make concurrent updates to the state files, leading to conflicts, data loss, and state file corruption.
- *Isolating state files.* When making changes to your infrastructure, it’s a best practice to isolate different environments. For example, when making a change in a testing or staging environment, you want to be sure that there is no way you can accidentally break production.

The above issues need to be addressed in-house when building the automation of the processes that make up the infrastructure. However, the main drawback of Terraform in the context of Cloud Agnostic automation is that it abstracts definitions in a poor way. In essence, Terraforms definitions are often a one-to-one rewriting of the properties issued by the cloud providers' native APIs. Thus, the only thing we gain relative to the native API is that the multi-cloud definition is given a common form of notation and a central tool responsible for orchestration. Terraform lacks proper abstraction mechanisms to hide implementation details by exposing a simple API.

A separate problem with Terraform is the poor support for deploying applications to a pre-created infrastructure. This is done by injecting initialization scripts onto the virtual machine. The script is usually written as a shell script, leading to a mix of declarative infrastructure definition and imperative initialization scripts.

7. Kubernetes cluster as resource orchestration and execution environment

The decision to choose a cloud provider is a very difficult one. On the one hand, the use of native solutions available from a given provider is very tempting due to the relatively high ease of implementation and the predictability of costs at the time of the decision. On the other hand, a strong attachment to a provider's specific solutions raises concerns about over-dependence, which may result in no easy path out in the future to an environment offering better value for money.

Many companies, for this reason, are opting for a certain compromise to loosen their strong ties to a single cloud, choosing the *Kubernetes* computing cluster environment as their primary runtime tool. The use of Kubernetes as a *Platform as a Service* (PaaS) provides a universal abstraction layer to build independence and loosens many of the strong ties to the native services of a given provider. Each of the major cloud service providers mentioned has Kubernetes cluster as a PaaS offer. In the AWS cloud, this is the *Elastic Kubernetes Service* (EKS), Microsoft provides it in the form of *Azure Kubernetes Service* (AKS) while Google provides it as *Google Kubernetes Service* (GKE).

Kubernetes cluster is a portable, extensible open-source software platform for managing tasks and services running in Docker containers. Most importantly, Kubernetes works with declarative configuration and automation expressed in YAML files called manifests. The state of the environment itself is maintained directly on the cluster itself, thus bypassing many of the problems we encounter when using Terraform. With the requirement to use cloud agnostic definitions, using Kubernetes as an abstraction layer separating us from direct interaction with the cloud is a very welcome solution. On the one hand, our application definitions have one and the same record regardless of the cloud on which the cluster is installed, and on the other hand, it is Kubernetes in the form of the relevant drivers provided by the operator that knows how to scale the demand for virtual machines (VMs) or other specific resources.

However, the problem arises when there is a need to provision resources that Kubernetes itself does not support, e.g. registering a sub domain, running a database, etc. Pure Kubernetes is mainly an execution environment where the orchestration of the necessary resources is severely limited. The following chapter presents a solution to this problem, which extends Kubernetes' capabilities theoretically in an unlimited way.

7.1. Extending Kubernetes functionality with Crossplane

The developers of Kubernetes have predicted the possibility of extending functionality through so-called *Custom Resources* (CR) (Kubernetes, 2023). Custom Resource is an extension to the Kubernetes API that is not necessarily available in the default Kubernetes installation. It represents a customization for a specific Kubernetes installation. However, many core Kubernetes features are now built using custom resources, making Kubernetes more modular (Kubernetes, 2023). CRs can appear and disappear in a running cluster through dynamic registration, and cluster administrators can update CRs independently of the cluster itself. A CR is simply customized structured data. In order to perform additional operations on it, there must be a process to enforce it. This process is the Custom Controller, which performs programmed actions based on the CR. Custom Controller is a specialized Kubernetes Pod, that is observing changes in CRs and respond accordingly to them.

The aforementioned functionality is the basis of the Crossplane tool (Crossplane, 2023). The purpose of Crossplane is to extend the Kubernetes cluster with the ability to provision any resources outside the cluster. This is all done using the same *YAML* manifests when configuring the environment. Crossplane provides extensions to Kubernetes Custom Resources, while also providing the corresponding Custom Controllers responsible for executing these definitions. The advantage of this solution lies in a unified way of deploying the application and instantiating the resources for that application. One common *YAML* manifest format combines both tasks into a single process. Previously described tools unified writing in only one of these areas: deployment or resource orchestration. Kubernetes with Crossplane combines both areas into one consistent mechanism based on *YAML* manifests. Let's take a look at the principles of Crossplane.

Crossplane introduces multiple building blocks that enable you to provision, compose, and consume infrastructure using the Kubernetes API. These individual concepts work together to allow for powerful separation of concern between different personas in an organization, meaning that each member of a team interacts with Crossplane at an appropriate level of abstraction.

The primary concept for extending the Kubernetes API is the *Composite Resource Definition* (XRD) (Crossplane, 2023). The purpose of the XRD is to define the details of the exposed API, which will then be used for resource provisioning. XRD provides the ability to define a cloud agnostic interface that will be translated into appropriate compositions. In order to be able to transform the XRD into specific resources, Crossplane provides the concept of *Composition*. This is an entity whose task is to define particular resource orchestration for given XRD. Composition is executed after the user provides proper *Claim* for particular XRD. Each XRD can have multiple Compositions, where each Composition can be responsible for handling different clouds. A Composition uses the appropriate *Providers* to perform the operation. Providers are implemented by open-source teams as well as by many companies,

including cloud providers. Often, Providers are using internally native APIs, like SDK or CLI prepared by cloud vendor. The task of the Provider is to expose the corresponding API and their execution mechanisms in the form of a Pod running in Kubernetes (Crossplane, 2023).

Let's look at an example in which we will build a Cloud Agnostic API for network provisioning across two clouds: AWS and Azure. Both clouds provide a very similar concept, however the implementation differs between the two. For example, Azure requires a *Resource Group* to be indicated for entities being created which is not the case in AWS. We want to encapsulate these differences in a single consistent definition of XRD. A basic, very simple example is shown in Table 3. The code on the left defines an API scheme for networking. It assumes the existence of three specific properties {region, addressSpace, subnetCidr}. The right-hand side of Table 3 shows an example of the *Claim* that is used to create a network by end user. *Claim* provides information from the user as to what environmental parameters he is interested in. It is an API prepared for the end user. All implementation details are not visible. In the presented example, the user indicates only three available settings.

Table 3.
Example XRD definition of API for subnet provisioning with corresponding Claim for a resource

```
# XRD definition API for networking
apiVersion: apiextensions.crossplane.io/v1
kind: CompositeResourceDefinition
metadata:
 name: xnetworks.example.com
spec:
 group: example.com
 names:
 kind: XNetwork
 plural: xnetworks
 versions:
 - name: v1alpha1
 served: true
 referenceable: true
 schema:
 openAPIV3Schema:
 type: object
 properties:
 spec:
 type: object
 properties:
 region:
 type: string
 addressSpace:
 type: string
 subnetCidr:
 type: string
```

```
# Claim for network
apiVersion: example.com/v1alpha1
kind: XNetwork
metadata:
 name: exampleNet
spec:
 compositionSelector:
 matchLabels:
 cloud: aws
 region: eu-central-1
 addressSpace: 10.40.0.0/16
 subnetCidr: 10.40.32.0/19
```

Source: own work.

The above *Claim*, shown in table 3 is executed by the corresponding *Composition*. Since the example supports two clouds then through the compositionSelector field inside the *Claim* we indicate which composite is to be used to create the resource. The example *Compositions* code is shown in table 4.

Table 4.
Example of two compositions for subnetwork XRD covering AWS and Azure clouds

# XRD definition API for networking apiVersion: apiextensions.crossplane.io/v1 kind: Composition metadata: name: azure.xnetworks.example.com labels: cloud: azure spec: compositeTypeRef: apiVersion: exaple.com/v1alpha1 kind: XNetwork resources: - name: resource-group base: apiVersion: azure.upbound.io/v1beta1 kind: ResourceGroup metadata: name: resource-group patches: - type: FromCompositeFieldPath fromFieldPath: spec.region toFieldPath: spec.forProvider.region - name: vnet base: apiVersion: network.azure.upbound.io/v1beta1 kind: VirtualNetwork spec: forProvider: resourceGroupNameSelector: matchControllerRef: true patches: - type: FromCompositeFieldPath fromFieldPath: spec.region toFieldPath: spec.forProvider.location - type: FromCompositeFieldPath fromFieldPath: spec.addressSpace toFieldPath: spec.forProvider.addressSpace[0] - name: subnet base: apiVersion: network.azure.upbound.io/v1beta1 kind: Subnet spec: forProvider: resourceGroupNameSelector: matchControllerRef: true virtualNetworkNameSelector: matchControllerRef: true patches: - type: FromCompositeFieldPath fromFieldPath: spec.subnetCidr toFieldPath: >- spec.forProvider.addressPrefixes[0]	*# Subnet Composition for Azure* apiVersion: apiextensions.crossplane.io/v1 kind: Composition metadata: name: aws.xnetworks.example.com labels: cloud: aws spec: compositeTypeRef: apiVersion: exaple.com/v1alpha1 kind: XNetwork resources: - name: vpc base: apiVersion: ec2.aws.crossplane.io/v1beta1 kind: VPC patches: - type: FromCompositeFieldPath fromFieldPath: spec.region toFieldPath: spec.forProvider.region - type: FromCompositeFieldPath fromFieldPath: spec.addressSpace toFieldPath: spec.forProvider.cidrBlock - name: subnet base: apiVersion: ec2.aws.crossplane.io/v1beta1 kind: Subnet spec: forProvider: vpcIdSelector: matchControllerRef: true patches: - type: FromCompositeFieldPath fromFieldPath: spec.region toFieldPath: spec.forProvider.region - type: FromCompositeFieldPath fromFieldPath: spec.subnetCidr toFieldPath: spec.forProvider.cidrBlock

Source: own work.

Note that the XRD hide the programming details of how to provision individual resources. The abstract XRD presented for networking is very concise and readable. Its implementation translates differently on different clouds. In the case of AWS, two resources will be created: *Virtual Private Network* (VPC) and *Subnet* inside the VPC. Azure requires three entities to achieve the same functionality. Firstly, we need to create a *Resource Group* (RG), in which we then place a *Virtual Network* (VNet) and one *Subnet*. All resources are listed in the resources section of the *Composition*. Parameter values supplied by the user from Claim are rewritten by the patches sections. The presented patches are a small sample of the possibilities offered by this mechanism.

The *Composite* itself indicates what type it implements in the compositeTypeRef field. Both compositions shown indicate the same XNetwork type. The final indication of which Composition is to be executed by the Crossplane is done by appropriately labelling it in the metadata section.

In figure 1 is presented a conceptual diagram showing the relationship between the Crossplain components. The DevOps team provides an XRD to the Kubernetes cluster describing the APIs available to end users and a set of Compositions that define in detail how the APIs are to be orchestrated. In the diagram from Fig. 1, the DevOps team has provided one XRD and two *Compositions*, using two different *Providers*, Azure and AWS. The respective Azure and AWS Providers were also previously installed on the cluster.

Consumer of a service deploys to Kubernetes *Claim* manifest with specification for requested service. In the example from Fig. 1, the *Claim* points to the AWS environment. Thus, the request will be handled by the respective *Provider*, which, based on the *Claim*, will provision the required resources in the AWS cloud. At the same time, the Composition provides information on which applications are to be installed in the Kubernetes cluster itself.

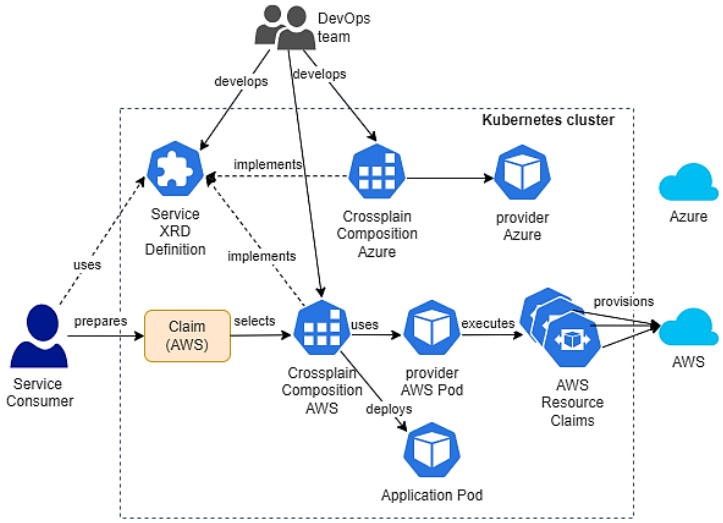

Figure 1. Diagram presenting the conceptual relations between Crossplain components in a Kubernetes cluster

Of course, we are in no way restricted to mixing resources from different clouds in a single *Composition*.

Each resource described in the *Composition* throughout its life cycle has a corresponding record on the Kubernetes cluster. This record stores the required state of the resource. This state is continuously monitored by Crossplane providers and, if differences are detected between the resource and its description in the cluster, the appropriate steps are executed. Through the Kubernetes cluster, Crossplane manages the entire life cycle of resources created in and outside the cluster. Removing *Claim* from the cluster also removes any resources created by it.

The example above illustrates how Crossplane extends Kubernetes functionality to create resources outside the cluster itself. This provides a uniform record of infrastructure definition and application deployment via *YAML* manifests. A state in which an API has been exposed that enables the application to run as a uniform record of the resource list and application deployment, e.g. in the form of a *Helm Chart Release*, is desirable. The orchestration of all elements is overseen by internal mechanisms that manage the lifecycle of Kubernetes objects. Thus, we gain a mechanism to prevent manual changes to the infrastructure, which is one of the requirements for well-designed automation of execution environments.

8. Conclusions

In the article, we provided an overview of the tools available to automate infrastructure. The IaaC problem is not an easy one to solve, particularly if you do not want to be strongly tied to a specific cloud provider. Achieving Cloud Agnostic status is much more difficult than automating within a single provider. In this case, it is not possible to design an effective automation process using the tools that the cloud provider provides. This is because these tools only work within a given provider, so we are forced to duplicate automation by specializing it based on different tools.

Tools that can automate across multiple cloud providers simultaneously may provide a solution to this problem. In particular, Terraform is a good solution. While Terraform provides a common format for declaring resources across multiple clouds simultaneously, it does not provide the ability to hide implementation details. In addition, Terraform was primarily developed for the purpose of automating infrastructure orchestration, and thus provides poor mechanisms for installing applications on the referenced infrastructure.

The most mature solution that meets the requirement for automation in isolation from the specifics of cloud providers' gives Kubernetes in combination with Crossplane. Pure Kubernetes successfully provides mechanisms for automating application deployment. In fact, it was primarily developed for such purposes. The only requirement to run a given application on a Kubernetes cluster is to package the application in an appropriate Docker container. Enriching Kubernetes with Crossplane extends the functionality of the cluster with the possibility of interacting with the external environment. Thus, we get a consistent, central

place where we manage the application as well as the infrastructure in a uniform way. All automation is written in the form of *YAML* manifests. The DevOps team simultaneously works on both infrastructure and deployment declarations, publishing the whole solution as a corresponding package. The definition itself is stored on the Kubernetes cluster providing a unified API. Deployment details are hidden behind the corresponding Compositions, providing the user only with a simplified XRD. At the moment presented solution seems to be most mature design when Cloud Agnosticism is key point on a list of requirements.

References

1. Azad, N., Hyrynsalmi, S. (2023). DevOps critical success factors — A systematic literature review. *Information and Software Technology, vol. 157,* Available online https://doi.org/10.1016/j.infsof.2023.107150, 15.04.2023.
2. *Azure Command Line Interface - overview* (2023). Retrieved from: https://learn.microsoft.com/en-us/cli/azure/, 15 April 2023.
3. *Azure/azure-sdk repository* (2023). Retrieved from: https://github.com/Azure/azure-sdk, 15 April 2023.
4. Brikman, Y. (2016). *How to manage Terraform state. A guide to file layout, isolation, and…* Gruntwork. Retrieved from: https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa, 15 April 2023.
5. CISO Report(2023). *Observability and security convergence*. Dynatrace 2023. Retrieved from: https://www.dynatrace.com/info/ciso-report/, 3 April 2023.
6. *Command Line Interface – AWS CLI* (2023). Retrieved from: https://aws.amazon.com/cli/, 15 April 2023.
7. Copado (2022). *Cloud Agnostic vs Cloud Native: Developing a Hybrid Approach*. Retrieved from: https://www.copado.com/devops-hub/blog/cloud-agnostic-vs-cloud-native-developing-a-hybrid-approach, 15.04.23.
8. *Crossplain Concepts* (2023). Retrieved from: https://docs.crossplane.io/v1.12/concepts/, 15 April 2023.
9. *Developer Tools - SDKs and Programming Toolkits for Building on AWS* (2023). Retrieved from: https://aws.amazon.com/developer/tools/, 15 April 2023.
10. *Google Cloud SDK - Libraries and Command Line Tools* (2023). Retrieved from: https://cloud.google.com/sdk, 15 April 2023.
11. *Install the gcloud CLI* (2023). Retrieved from: https://cloud.google.com/sdk/docs/install, 15 April 2023.
12. Kim, G., Humble, J., Debois, P., Willis, J. (2016). *The DevOps Handbook*. Portland: IT Revolution.

13. *Kubernetes Custom Resources* (2023). Retrieved from: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/, 15 April 2023.
14. Loschwitz, M. (2014). *Choosing between the leading open source configuration managers*. Admin Network & Security. Lawrence, KS, USA: Linux New Media USA LLC.
15. ScriptRock (2015). *Declarative v. Imperative Models for Configuration Management: Which Is Really Better?* Scriptrock.com. Archived from the original on 31 March 2015. Retrieved from: https://web.archive.org/web/20150331062438/https://www.scriptrock.com/blog/articles/declarative-vs.-imperative-models-for-configuration-management, 1.05.2023.