POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Executive Summary of the Thesis

Extending DevOps Modeling Language to support Kubernetes

Laurea Magistrale in Computer Science and Engineering

Author: Andrea Pirrotta

Advisor: Prof. Elisabetta Di Nitto

Academic year: 2023-2024

1. Introduction

The field of software development and engineering has seen big changes over the course of its life. One of the most significant change in recent years has been the shift toward cloud computing [15]. Using cloud computing, applications are hosted on infrastructures owned by a cloud provider and accessed through the internet. The pricing of cloud resources is very flexible, most commonly pay-as-you-go or dynamic [4]. This new computational model gives companies access to a virtually unlimited amount of computing, storage, and networking resources while avoiding the big initial cost in setup of an infrastructure and relieving them of the burden of infrastructure management.

1.1. DevOps and IaC

The switch from on-premise to cloud computing is, of course, not simple and comes with its own set of challenges. To fully exploit the power of the cloud, a new methodology called DevOps was invented. DevOps [6] integrates in a single large process the tasks of development and operation. A new flow of development, delivery, and monitoring is born, one that puts together provisioning and configuring of infrastructure, development of code, testing, deployment of software, and software monitoring. This unified process enhances cross-team collaboration, reducing software development and deployment time.
Central to the adoption of DevOps was the invention of Infrastructure as Code. IaC [9] is a solution that leverages domain-specific languages to describe and configure an infrastructure through code. Using code allows us to take advantage of all existing software development strategies and processes, improving collaboration and simplifying versioning. Infrastructure as code also makes the infrastructure easily replicable and testable. Since the infrastructure is in code form, we can use specialized platforms to automatically test it and deploy it, such automation is called CI/CD [12].
Particularly notable for the scope of this Thesis are the provisioning and configuration of infrastructure phases of DevOps. Provisioning [14] is the process of requesting infrastructural resources from a cloud provider. IaC tools such as Terraform [1] make the provisioning process easier and faster, allowing for fast and reliable replication and providing a single source of truth for the state of the infrastructure. Resource configuration [13] refers to configuring the state of Virtual Machines. The state includes installed software, running applications, and environment variables. IaC languages like Ansible [2] allow the

[1] Terraform website: `https://www.terraform.io`
[2] Ansible website: `https://www.ansible.com`

developer to easily configure the target machines by providing the steps that must be applied to correctly configure the VM's state. Complex configurations are broken down into small and simple tasks, easily reusable and idempotent.

1.2. Container Orchestration

The move toward cloud computing and the adoption of DevOps led to the design of a new software architecture: micro-services [5]. With micro-services, the functionalities of an application are broken down into its basic features, each developed and tested in isolation. The application then becomes an interconnection of services; invocation of a service happens through HTTP, RPC, or asynchronously by using queues and triggers. This architecture reduces the development and testing speed while making updates and the introduction of functionalities easier. The big downside to micro-services architectures is the big increase in platform management effort needed. A service, or a group of services, is deployed in a container, a lightweight application package deployed in a virtualized environment [10]. While using containerization enhances reliability and replicability of services, a new problem arises. Medium to large applications composed of many services require a large number of containers. These containers must be managed, monitored, replicated, and restarted or scaled when needed. Container orchestrators [7] such as Kubernetes [8] are the solution to this explosion in complexity. Kubernetes significantly decreases the containers' management overhead by allowing the user to define resources and their state declaratively. Kubernetes then takes care of keeping the state of the application consistent by spawning and restarting containers when needed, as well as automatically scaling the replicas of a container depending on some user-defined metric.

Container orchestration has become a necessity in modern service-oriented cloud applications, yet the design and setup of orchestrators such as Kubernetes is still complex and has a steep learning curve. There is no single DevOps tool that allows to easily provision and configure a Kubernetes cluster, and to define and deploy software on it. This process is usually handled by a series of CI/CD pipelines that are hard to build, harder to debug, and require knowledge of multiple DevOps technologies. We present the DevOps Modeling Language as a possible solution.

2. DOML

The DevOps Modeling Language [3] is a model-based language developed to support the DevOps practice and is part of the PIACERE Framework [11]. With DOML, a user can define the components of an application, model the infrastructure on which the application should be deployed, and map the application components onto the infrastructural components. This is achieved by means of the three layers that make up a DOML model: the Application Layer, containing the application components; the Abstract Infrastructural Layer, containing the abstract definition of the infrastructure components; and the Concrete Infrastructural Layer, containing the mapping of AIL components to a specific cloud provider's service. The Configuration part of a DOML model is used to map application components to AIL components. DOML is a unique tool thanks to its ability to initially separate application and infrastructure definition and to logically connect the two later, all within a single file. This, paired with the infrastructure abstraction of the AIL, makes DOML suitable for use by inexperienced users and offers a gentler learning curve compared to other IaC and DevOps tools.

A DOML model can then be translated into IaC such as Terraform and Ansible using the Infrastructural Code Generator [1]. The ICG is another component of the PIACERE Framework that is able to generate IaC from a DOML model. This is done using Template-Based Code Generation. For each target language, the ICG invokes the corresponding plugin. The plugin is then responsible for selecting the correct template for each DOML resource and populating it with information from the DOML model. This approach is very effective and easy to extend as we can write a new plugin to support another target IaC language or add new templates to existing plugins to support new resources.

DOML can be easily extended through the DOML-E extension mechanism [2], which allows the user to define new concepts and model them as new classes or as a specialization of existing classes. This mechanism, paired with the ICG

extension mechanism, makes DOML very flexible and easily extensible, allowing users to customize it for specific scenarios or environments.

2.1. Thesis objectives

The main objective of this Thesis is the extension of DOML, introducing Kubernetes into the language creating DOML-K8s. The need for container orchestration in modern cloud applications cannot be understated; providing support for Kubernetes ensures the usability of DOML for large cloud applications and brings DOML in line with similar technologies. The creation of DOML-K8s also serves as a validation of the extension mechanism of both the DOML and the ICG, further emphasizing the flexibility and extensibility of the language.

3. Implementing Kubernetes into DOML

The extension of DOML into DOML-K8s was a multi-step process. The first step is the analysis of Kubernetes, determining the elements that must be modeled into the DOML to support the technology. Once we have determined what we need to model, we move onto the actual extension, first extending DOML and then the ICG.
The most fundamental element to model is Kubernetes itself. This means modelling a Kubernetes Cluster, the environment in which all Kubernetes resources are deployed. A Kubernetes cluster is a collection of machines managed by the Kubernetes software. The cluster contains both the control plane, the node responsible for managing the entire cluster, as well as the worker nodes on which all resources are hosted. We decided to use EKS as our cluster, which is Amazon Web Service's managed Kubernetes service. To create an EKS [3] cluster, we simply indicate the EC2 instances that act as worker nodes in the cluster, and AWS handles the configuration and management of the entire Kubernetes infrastructure.
We can then move on to defining the Kubernetes resources, namely pods and services. A pod is the smallest element that can be addressed in Kubernetes. A pod represents a container or a group of containers; all containers of a pod are deployed on the same node and share the same IP address, thus they communicate through localhost. To replicate and attach resources to a pod, we use deployments. A deployment defines a replica set of pods, each replica is deployed on one of the available pods. To allow pods to communicate with the rest of the cluster and even outside the cluster, we use services. A service is attached to a pod through its deployment and assigns to the pod either an internal IP, used to address it from within the cluster, or an external IP that can be addressed from outside the cluster.
The final component to model is the Horizontal Pod Autoscaler(HPA), a component that is attached to a pod using a deployment. The HPA continuously monitors a user-defined metric for the deployment it is attached to; depending on the value of the metric, it can scale up or down the number of replicas of a pod. Modeling the HPA in DOML is fundamental to ensure the ability of the DOML users to define modern reactive applications, whose infrastructure can scale up or down depending on the workload, increasing the application's performance while minimizing cost.

3.1. DOML-K8s elements

Once we had identified what resources were to be modeled in DOML-K8s, the next step was the extension of DOML using DOML-E.
The EKS cluster was modeled using two AIL classes and one CIL class. In the AIL, we created two classes as extensions of the ExtInfrastructuralElement class: the KubernetesCluster defines the EKS cluster in terms of VPC and subnets in which the cluster is deployed, the security groups attached to the resource, and the worker nodes; the worker nodes were defined separately using the KubernetesNodeGroup class, that defines a group of EC2 instances through their instance type, as well as maximum, minimum, and desired size of the group. The CIL class is simply the KubernetesCluster concrete element that maps to the AIL cluster.
We then decided to abstract the definition of Kubernetes resources from the user as much as possible as a way of making the language easy to use and more general. For this reason, we did not explicitly model the pods and services but rather only modeled deployments. By declaring the image that is used in a deployment, we can

[3] `https://aws.amazon.com/eks/`

infer the definition of the pod, and by declaring the ports exposed by the image, we can infer the services attached to a pod. This means that we can use a single class to represent every Kubernetes resource in DOML-K8s. This class is called KubernetesDeployment; it is an extension of the Deployment class and maps an application component to a KubernetesCluster through a docker image. We can also attach an autoscaler to the deployment, as well as declare the number of replicas of the pod and a namespace for the resources. Unfortunately, the DOML did not contain a class suitable to represent a Docker Image and its exposed ports, so we also had to add two additional AIL classes, both extensions of the ExtInfrastructuralElement class: the DeploymentImage class represents a Docker image, it contains the URL for the image and is associated to the PodExposedPort class; the PodExposedPort class models a port exposed by a container, it has the container port, the exposed port, and can be specified as private or public so that the type of Kuberentes service can be inferred. We also needed to model the HPA, this was done using the KubernetesAutoScaler class, an extension of the ExtInfrastructuralElement with which the user defines the scaling metric, as well as max and min number of pods.
One final class we decided to add was the ExternalKubernetesDeployment, an extension of the deployment class through which the user can deploy an existing Kubernetes resource on the cluster. This was done to promote code reusability, make the language more usable for inexperienced users, and allow the usage of Custom Resources inside Kubernetes.

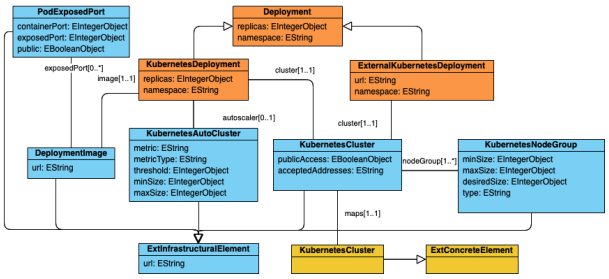

Figure 1: New DOML classes

A class diagram of the new classes and how they relate to one another can be seen in Figure 1.

3.2. ICG extension

The final step is extending the ICG to support the new classes introduced in DOML-K8s. Firstly, we must create a new Terraform template for EKS, this template is populated with the data from the KubernetesCluster and KubernetesNodeGroup classes. Then, we need to manage the generation of IaC for the Kubernetes resources. Kubernetes resources are defined using YAML Kubernetes files, thus we need to create a new Kubernetes plugin that handles the generation of the target IaC. This new plugin is invoked by the ICG when the parser identifies a resource with Kubernetes as the target language, which happens when parsing a KubernetesDeployment or ExternalKubernetesDeployment class. The plugin reads the data from the DOML-K8s model and populates the deployment template. The template contains the definition of the deployment, the pod, eventual services, and autoscalers. We do not need to create a template for the external deployment since it uses already existing Kubernetes YAML files. Finally, the plugin generates a file with the commands we must run to deploy all of our resources on the cluster.
Now that the ICG extension has been completed, we can generate the IaC to provision the EKS cluster and to deploy the Kubernetes resources on the cluster.

4. Validation

What is left to conclude the Thesis is to validate the ability of DOML-K8s to create Kubernetes applications. The validation was carried out by creating real-world applications that reflect how DOML-K8s would be used in a real enviroment. In particular, we want to prove that DOML-K8s can be used to provision and create an EKS cluster, to deploy multiple heterogenous containers on the cluster, to integrate multiple containers in an application, and to scale the application resources based on workload.

4.1. Scalable Web Server

For the first use case, we decided on a simple Scalable Web Server. The server is implemented through an Nginx web server and a Python backend and is packaged in a Docker image. The image is then deployed on the cluster through a deployment and is attached to a loadbalancer

service and an HPA that scales on the average CPU utilization. The rest of the infrastructure is very simple, comprising a single vpc with two subnets and the cluster. The following is a snippet of the Terraform code generated by the ICG and used to deploy the eks cluster:

```
module "eks" {
  ...
  cluster_name="Cluster_main"
  ...
  cluster_security_group_id =
  aws_security_group.sg_security_group.id
  vpc_id=aws_vpc.common_network.id
  subnet_ids=[aws_subnet.eks_subnet1_subnet.id,
  aws_subnet.eks_subnet2_subnet.id]
 ...
 eks_managed_node_groups = {
    main_group = {
      min_size     = 1
      max_size     = 3
      desired_size = 1
}}
```

The generated Terraform code for the EKS cluster contains all of the informations needed to correctly provision the resource, including: security groups, vpc and subnets, node group definition. The yml file for the Kubernetes resource was also generated by the ICG:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: sws-deployment
  ...
spec:
  replicas: 2
  ...
      containers:
      - name: webserver
        image: andreathesis/sws
...
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: sws-hpa
...
 scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: sws-deployment
 minReplicas: 2
 maxReplicas: 5
 metrics:
 - type: Resource
   resource:
    name: cpu
    target:
      type: Utilization
      averageUtilization: 50
```

In the snippet we can see how the ICG generated the deployment for the web server container, as well as its autoscaler. Since we used a loadbalancer service, we can access the website at its public address and stress test it to show the autoscaler in action.

```
NAME: sws-hpa
REFERENCE: Deployment/sws-deployment
TARGETS: cpu: 103%/50%
MINPODS: 2
MAXPODS: 5
REPLICAS: 2
AGE: 6m46s
```

Figure 2: HPA log

```
NAME                              READY  STATUS
sws-deployment-5bdd4c5645-9ksnz 1/1    Running
sws-deployment-5bdd4c5645-91vbx 1/1    Running
sws-deployment-5bdd4c5645-nwhdx 1/1    Running
sws-deployment-5bdd4c5645-psfn8 0/1    ContainerCreating
sws-deployment-5bdd4c5645-v6dj5 1/1    Running
```

Figure 3: Scaled resources

In Figure 2, we can see the HPA reading the utilization of the pods during the stress test. The HPA detects the CPU utilization to be above the threshold, set at 50%, so it will scale the current number of replicas, 2 in this case, to the maximum of 5. In Figure 3, we can see that the number of pods has now scaled up to 5, as we had defined in the KubernetesAutoScaler class of our DOML model.
With this first simple use case, we validated the ability of DOML-K8s to provision an EKS cluster, deploy resources on Kubernetes, and scale the resources automatically.

4.2. Distributed Scraping

For the second use case, we decided to create a more complex application: a Selenium Grid web scraper that, in order to prove the ability of DOML-K8s to integrate different components of an application, performs scraping on the website of use case one and scales on the number of queued jobs. A Selenium Grid is a distributed web scraping platform that uses a master-slave architecture. The master receives job (scraping) requests over HTTP and distributes the jobs to the worker nodes. The worker nodes read the required data from the website and return the result to the master node, which then responds

to the original client. Kubernetes is ideal for such distributed applications since we can scale the number of worker nodes according to the length of the job queue. To create our application using DOML-K8s, we must define a deployment for the selenium hub (master node) and the selenium nodes (worker nodes). The autoscaling part is the most complex one; since we need to scale on a custom metric, we should use a custom HPA and a custom metric server. We decided on the following architecture: a custom metric exporter pod that reads the queue length metric from the selenium hub and exposes it to the cluster; a Prometheus server that reads the queue length from the custom exporter; a Keda ScaledObject that scales the number of selenium nodes based on the queue length metric read from the Prometheus server. While the first component is deployed using a KubernetesDeployment, the other two components are external custom resources deployed using the ExternalKubernetesDeployment. The use of external IaC files may seem counterintuitive, but it is actually a very valuable feature of DOML-K8s. By using existing yml files, we allow code reuse, and make the use of complex application architectures possible for non expert users. These external resources are third party Kubernetes resources developed and open sourced by other expert developers, not giving users access to such resources and forcing them to build them from scratch would be a big limitation in using DOML-K8s.
The core of the DOML-K8s model is the same as the SWS use case and all of the resources are deployed on the same cluster as the Scalable Web Server. The deployment of software on the EKS cluster was handled using some of the commands generated by the ICG:

```
kubectl create namespace selenium
kubectl apply -f selenium-hub_deployment.yml
kubectl apply -f https://raw.githubusercontent
.com/andrimega/...
```

The commmands include deployment of resources generated by the ICG, as well as third party resources from the internet. Once all of the software has been deployed, we can send scraping requests to the grid over HTTP, receive the scraping results, and see the autoscaling in action.

```
NAME: horizontalpodautoscaler.autoscaling/keda-hpa-selenium-hpa
REFERENCE: Deployment/selenium-nodes-deployment
TARGETS: 0/5(avg)
MINPODS: 1
MAXPODS: 100
REPLICAS: 2
```

Figure 4: HPA log

```
NAME                                                    READY   STATUS
pod/selenium-grid-metric-exporter-deployment-56dd477d79-shgrn   1/1     Running
pod/selenium-hub-deployment-f75fc788f-c98pp             1/1     Running
pod/selenium-nodes-deployment-59b4c58775-4qt4f          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-4vmm8          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-5rz4w          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-9wwvr          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-c9tct          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-f2lzf          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-fzk51          1/1     Running
pod/selenium-nodes-deployment-59b4c58775-hm6j5          1/1     Running
```

Figure 5: Scaled resources

In Figure 4 we can see the HPA reading target metric from the pods. The metric is the queue size, which is 0 as we have yet to submit any scraping jobs. Since we have no jobs queued, only 2 replicas are online. As soon as we submit jobs, the HPA detects them and scales up the number of selenium node replicas. In Figure 3, we can see that the number of pods has now scaled up from the initial 2, multiple pods have been created as a result of the load on the system.
This application completes the validation of DOML-K8s, proving that DOML-K8s can be used to deploy complex Kubernetes applications composed of multiple heterogeneous containers. These applications can easily be integrated with other systems, as we have seen with the interaction between the grid and the web server. The application can also scale its resources according to custom user-defined metrics.

5. Conclusions

This Thesis has concluded with the successful extension of DOML into DOML-K8s. The addition of new classes to the DOML metamodel and new plugins and templates to the ICG has been concluded and thoroughly validated. The new language is able to describe and deploy complex Kubernetes applications composed of multiple components. DOML-K8s supports both the definition of Kubernetes resources and the use of external resources, promoting code reusability and increasing the overall usability of the language. Autoscalers can be attached to Kubernetes resources, allowing autoscaling on custom metrics.
DOML-K8s is robust and complete, yet it can still be extended and refined by users if needed, thanks to the elevated flexibility and extensibility of DOML. Future developments of the language could include the ingress services, the support for custom DNS records, or the addition of a custom resource into the language.

References

[1] Debora Benedetto, Lorenzo Blasi, and Laurentiu Niculut. D3.6 infrastructural code generation, aug 2023.

[2] Sergio Canzonieri. Exploring extensibility of devsecops modelling language (doml). Master's thesis, Politecnico di Milano, 2024. `https://hdl.handle.net/10589/223937`.

[3] Michele Chiari, Bin Xiang, Sergio Canzoneri, Galia Novakova Nedeltcheva, Elisabetta Di Nitto, Lorenzo Blasi, Debora Benedetto, Laurentiu Niculut, and Igor Škof. Doml: A new modeling approach to infrastructure-as-code. *Information Systems*, 125:102422, 2024.

[4] `https://www.binadox.com/blog/cloud-computing-pricing-models/`.

[5] Namiot Dmitry and Sneps-Sneppe Manfred. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

[6] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolás Serrano. Devops. *IEEE Software*, 33:94–100, 05 2016.

[7] Asif Khan. Key characteristics of a container orchestration platform to enable a modern application. *IEEE Cloud Computing*, 4(5):42–48, Sep. 2017.

[8] Marko Luksa. *Kubernetes in Action*. Manning Publications, 2017.

[9] Kief Morris. *Infrastructure as Code*. O'Reilly media inc., 2021.

[10] Claus Pahl, Antonio Brogi, Jacopo Soldani, and Pooyan Jamshidi. Cloud container technologies: A state-of-the-art review. *IEEE Transactions on Cloud Computing*, 7(3):677–692, July 2019.

[11] `https://piacere-project.eu`.

[12] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943, 2017.

[13] Nishant Kumar Singh, Sanjeev Thakur, Himanshu Chaurasiya, and Himanshu Nagdev. Automated provisioning of application in iaas cloud using ansible configuration management. In *2015 1st International Conference on Next Generation Computing Technologies (NGCT)*, pages 81–85, Sep. 2015.

[14] Sukhpal Singh and Inderveer Chana. Cloud resource provisioning: survey, status and future research directions. *Knowledge and Information Systems*, 49(3):1005–1069, Dec 2016.

[15] Jayachander Surbiryala and Chunming Rong. Cloud computing: History and overview. In *2019 IEEE Cloud Summit*, pages 1–7, Aug 2019.